UNIVERSITY OF OSLO
Department of Informatics

Distributed computing with the Cell Broadband Engine

Master thesis

Martin Wam

November, 2010

Distributed computing with the Cell Broadband Engine

Martin Wam

November, 2010

Contents

List of Figures

Abstract

The rapid improvements in the availability of commodity high-performance components has resulted in a proliferation of networked devices, making scalable computing clusters the standard platform for many high-performance and large-scale applications. However, the process of parallelizing applications for such distributed environments is a challenging task, requiring explicit management of concurrency and data locality. While there exists many frameworks and platforms to assist with this process, like Google's MapReduce, Microsoft's Dryad and Azure, Yahoo's Pig Latin programming language, and the Condor framework, they are usually targeted towards off-line batch processing of large quantities of data, contrary to real-time offloading of compute intensive tasks. Moreover, MapReduce, Dryad, and Pig Latin may not be suitable for all application domains, due to their inability to model branching and iterative algorithms.

In this thesis, we present a design for a framework able to accelerate applications by offloading compute intensive tasks to a heterogeneous distributed environment, and provide a prototype implementation for the Cell Broadband Engine. We evaluate the framework performance and scalability, and propose several future enhancements to further increase performance. Our results show that compute intensive applications that allow for high numbers of concurrent jobs fits well to our framework, and shows good scalability.

Acknowledgements

I would like to thank my supervisors Håvard Espeland, Pål Halvorsen and Carsten Griwodz for their guidance and inspiring attitude. Without their invaluable feedback, this work would not have been possible.

I would also like to thank the guys at the Simula lab for a great work environment, with great conversations and good laughs.

Last but not least, I would like to thank my family and girlfriend, for continued support, encouragement and endless motivation.

Oslo, November 8. 2010
Martin Wam

Nomenclature

DMA	Direct Memory Access
EA	Effective Address
EIB	Element Interconnect Bus
GPP	General Purpose Processor
IPC	Inter-process communication
LS	Local Store
LSA	Local Store Address
MARS	Multicore Application Runtime System
MFC	Memory Flow Controller
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
PPE	PowerPC Processor Element
PPU	PowerPC Processor Unit
SIMD	Single-instruction, multiple-data
SPE	Synergistic Processor Element
SPU	Synergistic Processor Unit

Chapter 1

Introduction

1.1 Background and motivation

From the introduction of the first general-purpose electronic computer in 1946, development of processor architectures has ushered in a constant change to satisfy the ever increasing demand for processing power. Intel's co-founder Gordon E. Moore published a paper in 1965, describing a trend where the number of components in integrated circuits had doubled every two years, and would continue to do so for the at least another decade [10]. This trend came to be known as Moore's Law, and has proven to be fairly accurate, as it has continued to hold true for more than half a century.

Due to the physical limitations of the transistor technology, further increases in processor clock frequency is heavily constrained by power consumption and heat generation, entailing the need for manufacturers to make architectural design changes. On a chip produced using AMD's 90mm process, power consumption increases by 60% with every 400 MHz rise in clock speed [11]. To keep fulfilling Moore's Law, manufacturers have shifted the focus from maximizing single-core performance, to increasing the number of processor cores per socket, known as multi-core architectures. However, the single threaded application does not typically benefit from these architectures, and may actually suffer from degraded performance, due to the lowered clock speed of each core. The shift towards new multi-core architectures poses several challenges to the application writer, mainly how the potential processing power can translate into an equal increase in computational performance. In addition to the cumbersome process of parallelizing an application, there is a limit to the performance gains a parallelized version of an application can achieve. These limitations are defined by Amdahl's Law, which states that a program cannot run faster than the aggregate of its sequential parts.

Furthermore, as the main improvements in performance of general-purpose processors have derived from higher clock frequencies and deeper pipelines, without a commensurate increase in memory speed, new bottlenecks emerge. Increasing the number of cores per socket results in augmented pressure on communication bandwidth, particularly with regard to memory access. In the last couple of decades, memory latency has increased by the several hundredfolds relative to clock speed, causing application performance in many cases to be dictated by latency, rather than in peak compute capability or peak bandwidth. This has motivated manufacturers to make radical design changes to the traditional architectures, introducing new, more specialized, heterogeneous designs.

Heterogeneity is not a new concept, as specialized co-processors have been used to supplement the functions of the main CPU since the early mainframe computers. The floating-point unit (FPU) became common as a co-processor already in the 1980s, to help speed up floating-point arithmetic, which at that point was done in software. Other applications of co-processors include network processors in Network Interface Cards (NICs), used to offload parts of the TCP/IP stack, dedicated Graphics Processing Units (GPUs) for image rendering, and Digital Signal Processors (DSPs) in sound cards. However, the availability of such components for general-purpose programming have until recent years been scarce, due to hardware having limited functionality, and restricted programming APIs. Today, heterogeneous architectures are widespread, designed with general-purpose computation in mind, combining processing elements with different characteristics for great amounts of processing power.

The Cell Broadband Engine [12] is a heterogeneous multi-core architecture, developed as a joint effort by Sony Computer Entertainment, Toshiba, and IBM in 2005. Although the architecture was initially intended for application in game consoles and media-rich consumer-electronics devices, the architecture has been designed to overcome the diminishing returns available from a frequency-oriented design point. A key optimization of the Cell Broadband Engine, is to provide the best combination of parallelism on all levels: The architecture features data level parallelism with both scalar, and single-instruction, multiple-data (SIMD) support. To exploit available memory bandwidth more efficiently, compute-transfer parallelism is provided by using programmable data transfer engines. Thread-level parallelism is supported with a hardware multithreaded PowerPC Processor Element (PPE), and multiple Synergistic Processor Elements (SPEs) on a single chip. In this effort, the Cell Broadband Engine bridges the gap between conventional multi-core architectures, and more specialized, high performance processors.

The rapid improvements in the availability of commodity high-performance components has resulted in a proliferation of networked devices, making scalable computing clusters the standard platform for many high-performance and large-scale applications. The main attractiveness of such systems is that they are built using affordable, low-cost, commodity hardware, fast local area networks (LANs), and standard software components such as Linux/UNIX, consequently leading to low-cost commodity super-computing. Where the demands of computationally bound problems exceed the computing resources available from a single machine, clustered environments allows for these problems to be solved in a timely and reasonable manner, by distributing the problem set amongst several nodes. However, writing code for non-trivial algorithms that dynamically scale to an arbitrary number of nodes is a challenging task, with regard to parallelizing the computation, explicitly handling concurrency and data locality, and non-deterministic behaviour.

To simplify the process of distributed computing, both industrial actors and open-source communities have contributed a significant body of research towards frameworks and programming models. Examples of these include Google's MapReduce [13], Microsoft's Dryad [14] and Azure [15], Yahoo's Pig Latin programming language [16], the Condor [17] system, and environments like PVM [18] and MPI [19]. However, most of these frameworks and models are targeted towards off-line batch processing of large quantities of data, and do not provide a lightweight interface for task offloading. There are also systems that adapt the MapReduce programming model, but target specific architectures only, like multi-core machines [20], GPUs [21], and the Cell Broadband Engine [22]. Moreover, MapReduce, Dryad and Pig Latin share a common drawback in their inability to model branching and iterative algorithms. While the Nornir [23] runtime system solves this latter problem by expressing algorithms in the Kahn process network (KPN) [24] model, it is targeted for shared-memory multi-core machines, and not distributed environments.

1.2 Problem Statement

Due to the complexity of parallel processing, with regard to managing concurrency, data locality, and non-deterministic behaviour, frameworks and programming models have emerged from industrial actors and open-source communities, to help simplify the process of parallelization on both standalone machines and in distributed environments. However, most of the frameworks targeted for distributed computing are designed for off-line batch processing and analysis of vast amounts of data, and do not provide a simple interface for offloading compute intensive tasks. We want to investigate how to make an efficient runtime system that allows for a simple offload-model, exploiting the properties of a heterogeneous distributed environment. Among the issues we investigate in this thesis, are the design challenges posed by such a framework, the work distribution strategies required by applications for the integration and adaption to an offload-model, and the potential scalability it may provide.

1.3 Main Contributions

In this thesis, we have presented a design and a prototype framework that can be used to accelerate applications, by offloading compute intensive parts of a program to a heterogeneous distributed environment. While we intend for this framework to support a broad range of platforms, we have due to time constraints limited the scope of this thesis to a prototype implementation for the Cell Broadband Engine, and leave the extensions to additional architectures as further work.

With the increasing number of cores in newer processor architectures, and with the proliferation of networked commodity components, application writers should be able to reap the benefits in accordance to Moore's Law, without the need to re-write or re-think the whole application. Therefore, we investigate work distribution strategies for various applications, and evaluate their potential scalability with added hardware. To achieve this, we implement a Motion JPEG video encoder using our framework, then adapt an existing implementation of the Advanced Encryption Standard (AES). These implementations have very different characteristics, which will reflect in both scalability and throughput.

For the Cell Broadband Engine, we have proposed a prototype of a workload scheduler, to be used in conjunction with a runtime library. We evaluate the performance of the servers with regard to workload processing, and propose programming models for the Cell Broadband Engine architecture.

1.4 Outline

The rest of this thesis is organized as follows; In chapter 2, we look at the technical details of multi-core architectures, in particular the Cell Broadband Engine, and cover parallelization techniques and parallel programming models. In chapter 3, we discuss the design challenges of such a framework, and propose a prototype design for the Cell Broadband Engine. The design proposed in chapter 3 is then realized in chapter 4, which covers the implementation details. In chapter 5 we evaluate the framework with video encoding, and in chapter 6, we use our framework to implement block cipher encryption. We then discuss the results and lessons learned in chapter 7, and summarize and conclude our thesis, as well as propose further work, in chapter 8.

Chapter 2

Background

In this chapter, we introduce the technical aspects of multi-core architectures, in particular the Cell Broadband Engine. Further, we go into the details and challenges of parallel programming, and introduce parallel programming models. We also give insight to current frameworks for distributed computing, and frameworks targeted for heterogeneous architectures.

2.1 Introduction

While we are still witnessing Moore's Law by the production of higher density processors, we have reached a plateau on single core performance. As the processor density increases, we get more processing power, along with higher power consumption and heat generation. In traditional architectures, heat generated by each new generation of processors have increased at a greater rate than clock speed. These posing power and thermal constraints have lead to the design of multi-core architectures, integrating two or more processor cores into a single processor socket. Thus, we get increased potential computational performance, but with less power consumption and heat generation. Though, the shift towards multi-core architectures poses several new challenges for both computer architects and programmers.

Providing an increase in computational performance that scales with the number of processor cores, is a challenge both when it comes to the design of the architecture, and its programmability. As processors increase in speed, a key challenge is reducing memory latency, as well as scaling up memory bandwidth in proportion with the increase in computational performance. Parallelizing and optimizing code for multi-core systems, synchronizing tasks, and the resource management that comes with it, may often be a complex and tedious task. Thus, having reliant tools and frameworks to reduce the programming effort is necessary.

2.2 Multi-core architectures

Multi-core architectures places two or more independent processing units (cores) on a single integrated circuit die, or onto multiple dies in a single chip package. Todays multi-core architectures are in many ways much like the older shared memory systems, as they merely migrate the shared memory system onto a single chip. There are several design approaches to these architectures, each with their own trade-offs and benefits, which is discussed later. By changing the level of integration and how resources are shared or partitioned, each design presents a different set of pros and cons. Figure 2.1 shows three different ways of implementing multi-core architectures; Shared cache and I/O interface, private cache with shared I/O interface, and private caches and private I/O interfaces on two dices. Note that cache in these designs represents higher levels of cache (L2 or L3), as L1 cache is too tightly integrated and is considered part of the core. Although there are many ways of implementing multi-core designs, there are two main categories — homogeneous and heterogeneous. These are covered in detail in section 2.2.2.

2.2.1 Multi-core design approaches

The shared cache approach has the advantage of having the lowest communication latency between cores. Data traffic between the cores does not have to go via the I/O interface, thus maximizing bandwidth to other devices, and letting the I/O interface focus on off-chip communication. Further, a shared cache approach lets the cache be dynamically shared between the cores. It only requires a single bus or network drop, making the interconnect network between the sockets simpler. However, while having high performance, it is one of the more complex designs. The cache controller needs to manage sharing policy and dynamic allocation between the cores, which might lead to performance issues if one of the cores is occupying all resources. Also, the cache needs higher bandwidth, as it serves two cores. The Intel Core Duo is an example of a

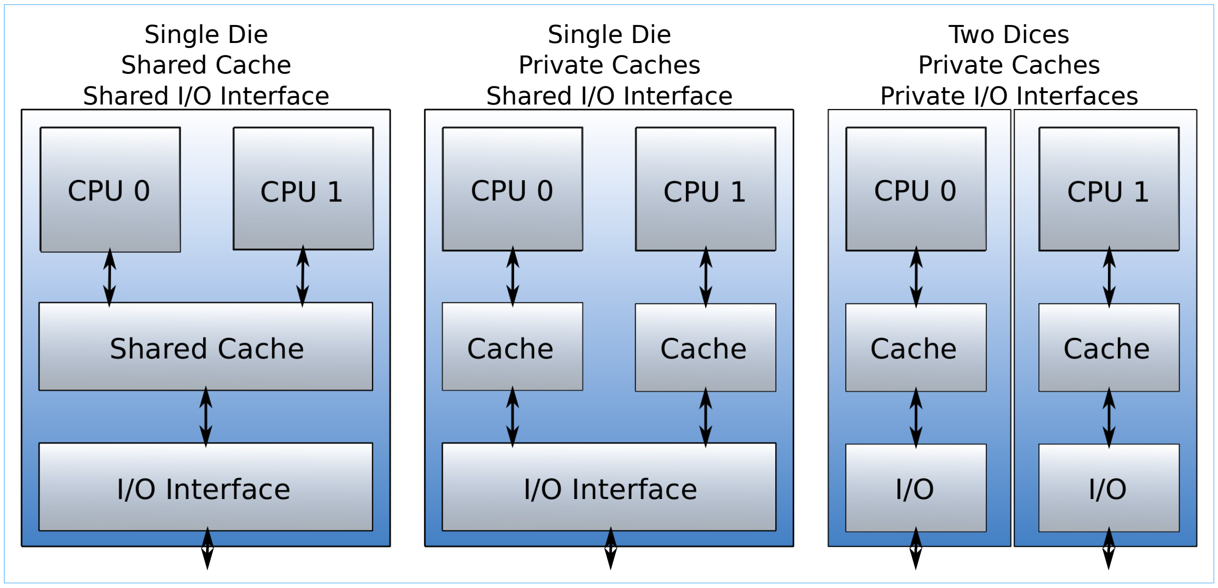

Figure 2.1: Three different multi-core designs [1].

processor that implements the shared cache design.

The shared I/O interface architecture has the advantage of having a simpler design, but lacks some of the flexibility and performance of a shared cache architecture. Like having a shared cache, communication between the cores does not have to go outside the chip, and the bus controller should be able to handle off- and intra-chip traffic simultaneously. Since the caches are separate, the cache controller may be identical to single core microprocessor units (MPUs), and relatively few modifications are needed to produce shared interface chips. However, since the caches are not shared, there can be no dynamic cache allocation between the cores, which may lead to wasted resources. Montecito is the code-name of a release of Intel's Itanium 2 processor family, which implements the shared interface design.

The shared package approach, often called Dual Chip Module (DCM) or Multi Chip Module (MCM), has the advantage of being less complex, as it does not require modifications to the CPU logic. Each die may also be tested and verified before packaging, granting yields close to that of a single core. The greatest disadvantage with this design, is the communication between the cores, being just as slow as communication between two sockets in a Symmetric Multi-Processor (SMP) approach. The lack of integration greatly increases data traffic across the external interface, and puts a higher load on the bus or interconnect network. Intel's Pentium D, code-named Presler, is an architecture that implements the shared package design [1].

2.2.2 Homogeneous and Heterogeneous Multi-Core Architectures

Homogeneous multi-core architectures are the more common architectures in today's workstation and desktop computers, consisting of one core design repeated consistently; All cores have the same frequency, cache size, functions, memory model, and so on, which makes it easier to produce. Also, since the cores are identical, it allows for a simpler programming model. However, with heterogeneous architectures, each core could have a specialized function. Heterogeneous architectures usually consists of a general purpose processor (GPP), and one or more specialized cores, often with a reduced instruction set, to improve performance on certain tasks or functions. There is an apparent trade-off between the architectures in terms of complexity and customization; Heterogeneous systems could have a centralized core for generic processing and running an operating system, a core for graphics, a core for audio, a heavily math-oriented core, and a core for communication based tasks. Though this is a more complex system, it may have efficiency, power, and thermal benefits that outweighs the complexity. The Cell Broadband Engine is an example of a heterogeneous architecture, and is introduced in detail in section 2.3.

Although the average x86 architecture is not considered as heterogeneous, it does in fact provide heterogeneity on several levels; With NVIDIA's parallel computing architecture CUDA [25], GPUs are becoming increasingly more available to general purpose programming, making virtually all desktop computers heterogeneous systems. In addition to the FPU, Intel has later added extensions to their instruction set architecture, to include technologies like Streaming SIMD Extensions (SSE) [26], and AES New Instructions (AES-NI) [27]. However, without compilers and runtimes to support these technologies, it is still up to the application writers to exploit the potential processing power the system offer.

2.2.3 Multi-core challenges

There are several challenges that needs to be considered with a multi-core architecture. Adding an additional core to the architecture does not necessarily mean a hundred procent performance increase. One also needs to consider the overhead associated with interprocessor communication, shared resources and diminishing returns. Distributed caches raises the challenge of cache coherence, and additional cores may also rise problems with power and thermal management.

Cache coherence

Cache coherence is a challenge with multi-core architectures when having distributed L1 and L2 caches. Since each core has its own set of caches, the current version of data may not always be the one most up-to-date.

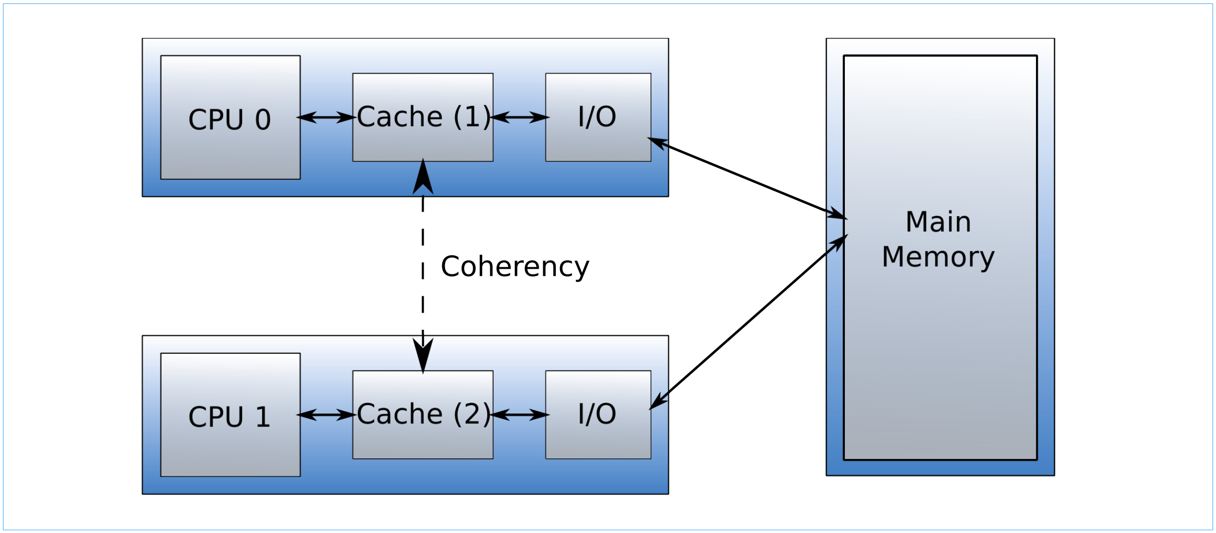

Figure 2.2: Distributed caches and cache coherence.

Consider the scenario in figure 2.2, where both processors read a block of memory into its private cache. CPU 0 then writes a specific value to that memory location, making the data in cache one and two inconsistent. When CPU 1 attempts to read the value from its cache it does not have the updated value, unless the cache entry is invalidated and a cache miss occurs. The cache miss then forces the entry in cache two to update. If certain mechanisms like this is not enforced, it results in incorrect data and possibly program crashes.

There are several mechanisms to obtain coherency between caches, each having their benefits and drawbacks. The two most common techniques are directory-based coherence, and snooping. Directory-based coherence places the shared data in a common directory, that maintains the coherence between caches. This directory acts as a filter, as the processor needs to ask for permission to load an entry from primary memory to its cache. When an entry changes, the directory either updates or invalidates the other caches with that entry.

The snooping protocol only works on bus-based systems, as each cache uses the bus to monitor memory access to locations they have cached. When a write operation is observed to a location currently cached, the cache controller invalidates its own copy of the snooped memory location. Although snooping uses more bandwidth, since all cores needs to monitor all requests and responses, it tends to be faster than directories. However, snooping does not scale well, as each added core requires more bandwidth and a larger bus. Directory messages are point to point, uses less bandwidth than snooping, and may therefore be more suitable for systems with a large number of cores.

Power and thermal management

Adding an additional core to the chip, in theory, would make the chip consume twice the amount of power, and generate an additional large amount of heat. As mention earlier, the clock frequency of each core is therefore lowered to compensate for the heat generation and power consumption. Many designs also incorporate a power control unit, which is able to shut down unused cores. By powering off cores and using clock gating, which disables portions of a circuit so that its flip-flops do not change state, one is able to reduce the amount of leakage in the chip.

Further, another common power management technique is dynamic voltage scaling, where the voltage to the chip is either increased or decreased, depending on the circumstances. Increasing the voltage is known as overvolting, contrary to undervolting, where voltage is decreased. Undervolting is done when the temperature of the core needs to be reduced, and when conserving power consumption, which is particularly important for laptops and mobile devices. When a performance increase is needed, overvolting is done to apply a higher voltage to the circuit, allowing capacitances to charge and discharge faster. This results in faster operation of the circuit, allowing a higher frequency of operations. Dynamic voltage scaling is in most cases accessible through the BIOS. However, when not accessible through software, hardware modifications are needed.

Dynamic frequency scaling is a technique which is closely related to dynamic voltage scaling. This power management technique is often used in conjunction with dynamic voltage scaling, since the maximum frequency a chip may run at is related to the operating voltage of the chip. Most modern processors are strongly optimized for low power idle states, making the conjunction a necessity to optimize power and thermal management. Implementations of dynamic frequency and voltage scaling includes Intel's *SpeedStep* [28] series for the mobile processor line, and AMD's *Cool'n'Quiet* [29] and *PowerNow!* [30] processor throttling technology.

Reducing the clock speed and voltage of the processor does indeed reduce heat generation. However, it should be noted that doing this may not reduce power consumption. As the technologies mentioned above can reduce power consumption when the processor is in an idle state, we obviously want the processor to stay idle for as long as possible. This means executing the required tasks as quickly as possible, so the processor can return to idle state faster. Switching to the highest voltage and clock speed, executing the code, then dropping back to idle state, may actually save power consumption.

2.3 The Cell Broadband Engine

The Cell Broadband Engine, commonly abbreviated Cell, Cell BE, or CBEA, is a heterogeneous multi-core processor developed by "STI", a joint effort by Sony Computer Entertainment, Toshiba, and IBM, which formed in mid-2000. STI's objective was to develop a processor able to achieve 100 times the processing power of the PlayStation 2 [31]. The STI design center opened in 2001, and the Cell BE was developed over a period of four years. On the 17th of May, Sony Computer Entertainment confirmed that the PlayStation 3 (PS3) would be shipped with the Cell BE, with the possibility of installing third party software [32]. In August 2009, it was announced that the Cell processor in the newer PS3 Slim had moved from 65nm to the 45nm version. However, the Slim-version would unfortunately not have the possibility to install a third party operating system [33]. On April 1, 2010, Sony released firmware update v3.21, rendering all PS3s unable to install other operating systems [34]. Although the upgrade is optional, choosing not to install it disables features like playback of copyright-protected videos over DTCP-IP, and access to the PlayStation Network.

In 2008, IBM announced the release of a revised version of the Cell BE; the PowerXCell 8i. This newer version had improved double-precision floating-point performance on the SPEs, in addition allowing up to 32GB of slotted DDR2 memory. The PowerXCell 8i is available in IBMs QS22 Blade Servers, and is also used in IBMs Roadrunner supercomputer, which is the first supercomputer able to achieve the high-performance computing goal of one petaflop [35].

2.3.1 Hardware overview

The Cell BE architecture extends the 64-bit PowerPC architecture host processor (PPE) with cooperative synergistic offload processors (SPEs), supporting SIMD-operations for single and double point precision instructions. Connecting the PPE with the SPEs and I/O-elements, there is a specialized high-bandwidth circular data bus called the Element Interconnect Bus (EIB). The PPE and SPEs use fully cache coherent direct memory access (DMA) over the EIB to transfer data between main memory, local store (LS) and other external storage devices. Each of the SPEs have their own DMA engine to fully utilize the potential asynchronous and concurrent processing [31].

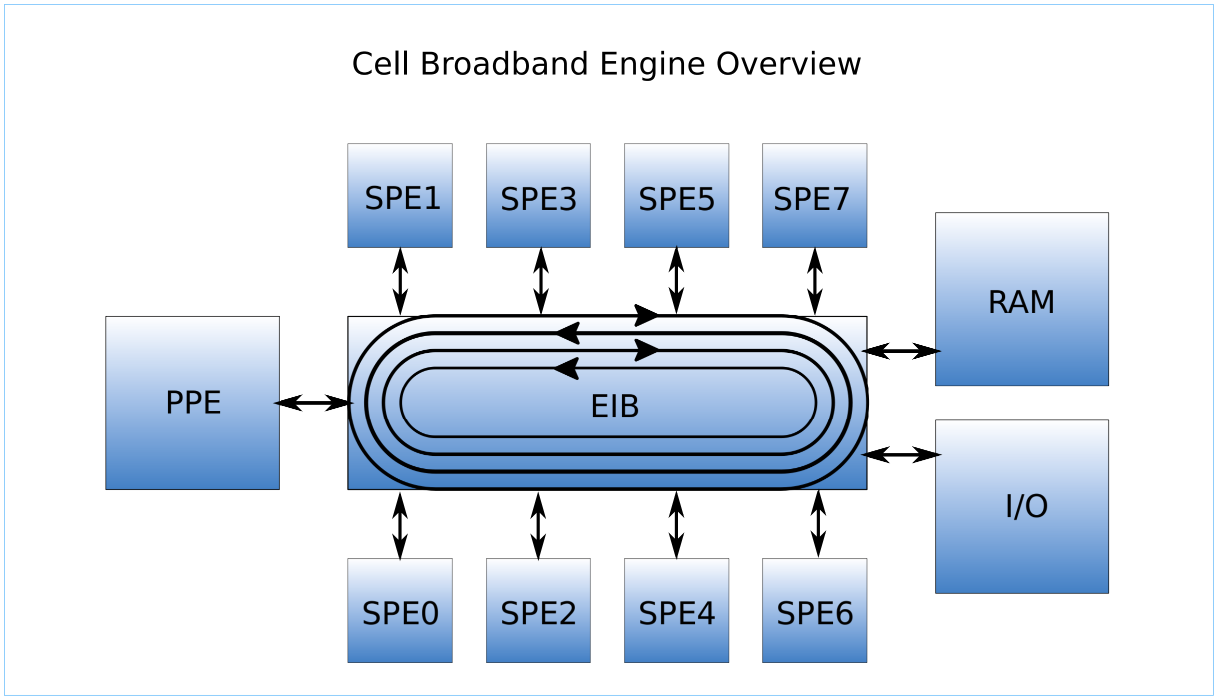

Figure 2.3: Cell Broadband Engine Architecture Overview.

The function of the processor elements is specialized into two types; The PPE is optimized for control tasks, and the eight SPEs provide an execution environment optimized for data processing [36]. With a clock speed of 3.2 GHz, the Cell BE has a theoretical peak computational performance of 230.4 GFlops per second (single-precision floating point) [12].

Power Processor Element (PPE)

The PPE is a dual-issue, dual-threaded, and in-order 64-bit Power-architecture connected to a 512KB L2 cache and 32KB L1 cache. The PPE is the main processor (host) of the Cell BE, and is responsible for running the operating system and control tasks [37]. The implementation of the PPE in the Cell BE includes innovations of Power such as virtualization and support for large page sizes, and provides compatibility with Power which enables the possibility to run conventional operating systems, like Linux. An overview of the PPE is shown in figure 2.4.

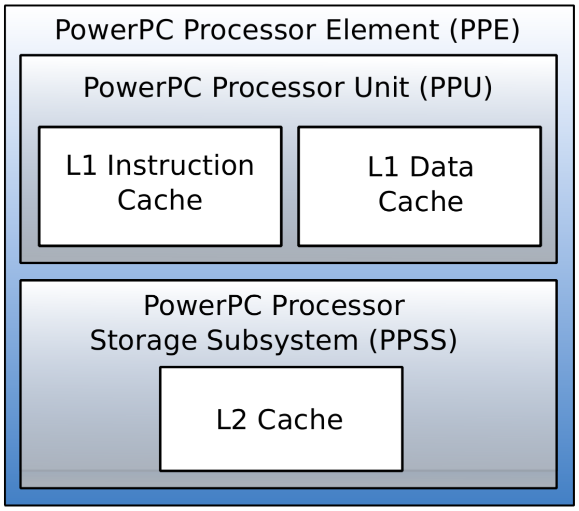
PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

Figure 2.4: Cell BE PPE block diagram.

Synergistic Processor Element (SPE)

One of the key architectural features of the Cell BE is the SPE. The SPE is a RISC processor with 128-bit SIMD-organization for single and double precision instructions, and is optimized for computation-intensive applications. The SPE is an independent processor, able to run an independent application thread. It is composed of a Synergistic Processor Unit (SPU), with a Memory Flow Controller (MFC) which contains a DMA, MMU and bus interface, enabling it to have full access to coherent shared memory, including memory-mapped I/O-space. It also contains 256KB embedded SRAM called

Local Store (LS), which can be addressed by the PPE through software, and is used for efficient instruction and data access by the SPU. However, LS must not be mistaken for conventional cache, as it is not transparent; It requires explicit DMA transfers to load and store data. An overview of the SPE is shown in figure 2.5.

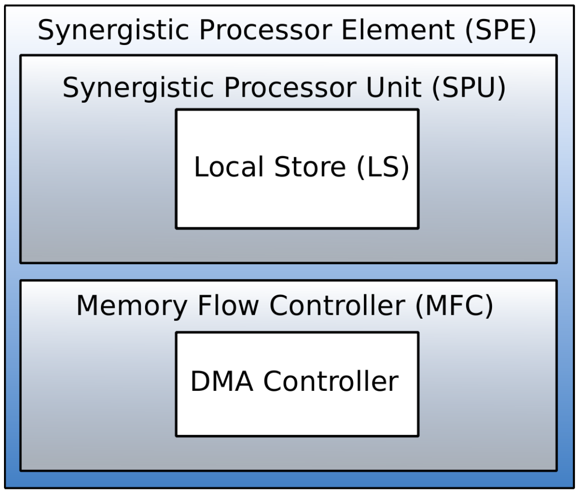
Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
Local Store (LS)
Memory Flow Controller (MFC)
DMA Controller

Figure 2.5: Cell SPE block diagram.

Element Interconnect Bus (EIB)

The Element Interconnect Bus enables internal communication between the PPE, SPE, main memory and external I/O. The EIB is implemented as a circular ring, comprised of four 16 byte wide unidirectional channels, which counter-rotate in pairs. At maximum concurrency, each channels allows up to three transactions. Each unit connected to the EIB has two 16 byte ports available — one for sending, and one for receiving. These can be accessed simultaneously, and sends and receives every bus cycle. The data rings are illustrated in figure 2.3.

The reason for having the rings run both ways, is to optimize the traffic patterns when units are communicating. The theoretical peak bandwidth for the EIB is 204.8GB/s at 3.2Ghz (128Bx1.6GHz). However, if all the units are trying to either send or fetch data from main memory concurrently, traffic patterns may be limited, resulting in idling on two of the paths. This is also the case for communication between the SPEs, with regard to the relative position between source and destination [38].

Memory Flow Controller (MFC)

The Memory Flow Controller is part of each SPE, creating an interface for the SPE to be used by main storage, processor elements, and other system devices, by means of the EIB. It provides the SPE with data transfer and synchronization capabilities, and also offers storage protection on the main-storage side of DMA transfers. Software running on the SPEs communicate with the PPE, main storage, and other SPE devices through channels. Channels are unidirectional message-passing interfaces, supporting 32-bit messages and commands. These commands may be used to initiate DMA transfers, query DMA status, perform MFC synchronization, inter-process communication via mailboxes or signal-notification, and so forth.

2.3.2 Overcoming the limitations

As mentioned is section 2.2.3, multi-core architectures rise many issues and challenges that needs to be overcome for it to be efficient. By optimizing control-plane and data-plane processors individually, the Cell BE overcomes the challenges regarding power, heat and memory management, providing approximately a ten-fold the peak performance of a conventional processor on the same power budget [39].

Power limitations

To significantly increase the performance of multi-core architectures, power efficiency has to improve at about the same rate as the computational performance. One way of doing this is to differentiate processors on what kind of jobs they are optimized to do, which ensures good utilization efficiency of the available resources.

The Cell BE differentiates its processors this way, by providing a general purpose PPE to run the operating system and other control-intensive tasks, as well as eight specialized synergistic SPEs for computational-heavy applications. The SPEs have a simpler hardware implementation, and do not devote transistors to branch prediction, out-of-order execution, register renaming, and extensive pipeline interlocks. Thus, more transistors are used for computation than on a conventional processor core.

Memory limitations

Processors have increased in speed faster than memory latency has improved, resulting in program performance being dictated by the activity of moving data between main memory and the processor. On symmetric multiprocessor platforms in the multi-gigahertz range, latency to DRAM memory is approaching 1000 cycles [40]. Hardware cache mechanisms are supposed to relieve the programmer of these tasks, but it has become increasingly more common for programmers and compilers to deal with this explicitly.

To deal with main memory latency, the Cell BE uses a three-level memory structure, with main memory and local storage, in addition to a large register file for each of the eight SPEs. The Cell BE also makes use of asynchronous DMA transfers between main memory and local storage, allowing up to 128 simultaneous data and code transfers. These features allows for the programmer to more efficiently schedule loads and transfers in a highly parallel environment.

Frequency limitations

To achieve higher operating frequencies, conventional processors require increasingly deeper instruction pipelines, to the point where it is suffering diminishing returns, and even negative returns if power is taken into account.

By specializing the PPE and the SPEs for control- and compute-intensive tasks, respectively, both the PPE and SPEs can be designed for high frequency without excessive overhead. The PPE achieves efficiency by having hardware multi-threading, rather than optimizing for single-thread performance. The SPEs achieve efficiency by having a large register file, and access to DMA transfers, supporting many concurrent memory operations.

2.3.3 Inter-process communication

Since the Cell BE is not a traditional shared-memory multiprocessor, other means are necessary to provide inter-process communication (IPC). The SPEs can execute programs, and directly load and store data from its local store. The system is therefore provided with three primary communication mechanisms: DMA transfers, mailbox messaging and signal-notification messaging. All three mechanisms are controlled by the SPEs memory flow controller (MFC).

DMA transfers

The Cell BE's DMA engine is part of the MFC unit, and is responsible for moving data within the Cell BE and external I/O devices. Each SPE has one dedicated DMA engine, with a corresponding control- and memory management unit (MMU) in the MFC. The MMU allows for moving streaming data in and out of LS in parallel with code execution on the SPE, which in turn allows for higher throughput. The DMA control unit consists of two queues - one for PPE-initialized requests, and one for SPE-initialized requests. The SPE uses DMA channels to initiate the DMA transactions, where each individual channel maps to a DMA parameter. These parameters include Local Store Address (LSA) and Effective Address (EA) to initiate transfers between the two storage domains, the size of the data to be transferred, a tag to identify the DMA or group of DMAs, and the MFC DMA command. Regarding alignment, DMA transfers must be 1, 2, 4, 8, or a multiple of 16 bytes, up to a maximum of 16kb, and aligned on a 16 byte boundary.

Most MFC commands have names that imply the direction of the transfer, and is always referenced from the perspective of the SPU that is associated with the MFC command. Thus, *GET* moves data from main memory to LS, and *PUT* moves data from LS to main memory. The other group of MFC commands are synchronization commands, used to control the order in which storage access is performed. The tag associated with the MFC command is a 5-bit tag group ID. SPE programs can use this tag ID to check on the status of a tag group, enable an interrupt when one or more tag groups are complete, or to enforce an execution order of commands within a tag group.

Mailbox messaging

Mailboxes on the Cell BE are special purpose registers in the SPEs and PPE, supporting sending and receiving of 32-bit messages. Each SPE has access to three mailbox channels, each of which are connected to the SPU's MFC. Two of the mailboxes are outbound, where one can send interrupts, and are used for sending messages from the SPE to the PPE. These mailboxes are limited to one entry each. The third mailbox is inbound, and can hold up to four entries. The inbound mailbox acts as a FIFO queue, where the oldest message is read first. If a program on the PPE would try to write more than four messages to the queue before the SPE program reads the data, the last message in the queue is always replaced, and the previous message is lost.

It should be noted that mailbox operations from the SPE are blocking operations. An attempt to write to a full outbound mailbox channel stalls the SPE program until the mailbox is read by the PPE. Similarly, a read from an empty inbound mailbox also stalls the SPE. However, this can be avoided using a read-channel-count instruction, checking the message count on the corresponding mailbox before deciding to do a read or write. Stalling does not occur on the PPE. Although mailboxes are intended to communicate messages such as program status and completion flags, they may however also be used for sending other short data, like storage addresses, function parameters or command parameters.

Signal-notification messaging

SPE signal-notification channels are connected to the SPEs inbound registers, and may be used to send information such as completion flags or program status from the PPE or other SPEs. Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register that can be written with signal-notification data. The MMIO registers are contained in the SPE's MFC.

Upon reading a signal-notification channel from the SPE, hardware clears the channel atomically. SPE software can use polling or blocking when waiting for a signal, or set up interrupt routines to catch the signals as they appear asynchronously. However, reading from the main-storage's MMIO does not clear its signal-notification register. Each SPU has two signal-notification channels, one to read each of the MMIO registers, where each channel is 32 bits long.

2.3.4 Bit ordering and bit numbering

Storage of data and instructions in the Cell BE uses big-endian ordering, which has the characteristics of storing the most-significant byte at the lowest address, and the least-significant byte stored at the highest address. Bit numbering within a byte goes from the most-significant bit (bit zero) to the least-significant bit (bit n). A summary of the byte-ordering and bit-ordering in memory and the bit-numbering conventions is shown in figure 2.6.

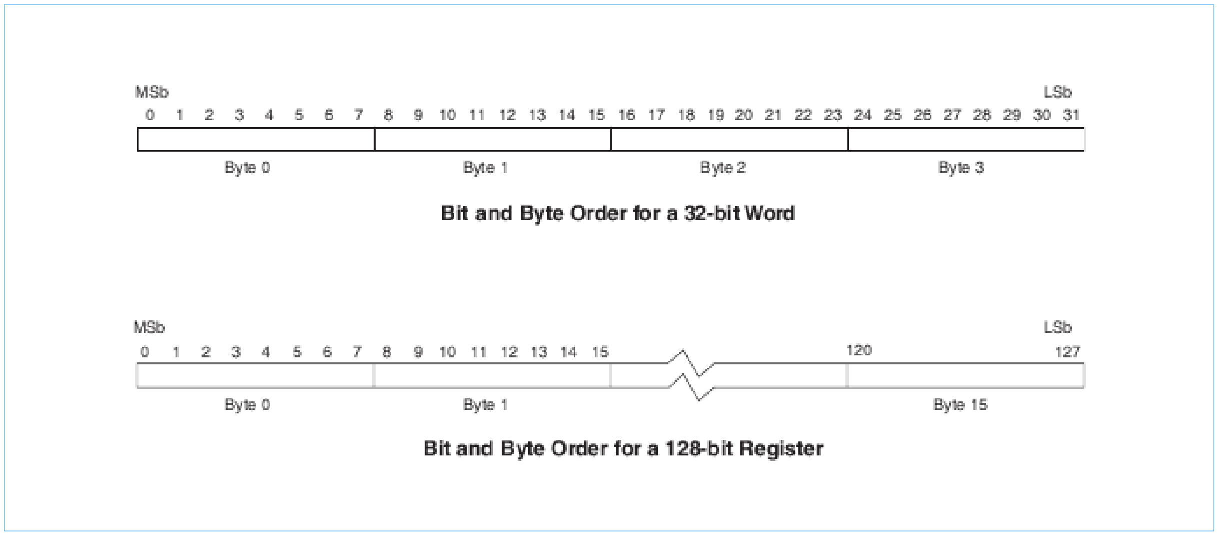

Figure 2.6: Big-Endian byte and bit ordering. Figure taken from the Cell Broadband Engine Programming Handbook [2].

Both the PPE and SPEs, including the MFCs, use big-endian byte-ordering. However, as DMA transfers are simple byte moves without regard to the numeric significance of the byte, byte-ordering becomes irrelevant to the actual movement of a data block. Byte-ordering only becomes relevant when data is loaded and interpreted by a processor element or the MFCs.

2.3.5 PlayStation 3 and Linux

The PS3 contains the first commercial application of the Cell BE. The processor is clocked to 3.2GHz with a 512KB L2 cache, and has six accessible SPEs. This is due to the seventh being dedicated to the OS for security aspects, and the eight due to production yields. Access to the RSX GPU is also restricted by a hypervisor implemented by Sony. Since the launch of the PS3, the Cell has had its process reduced from 90nm to 45nm. The latter is shipped with the PS3 Slim edition, which does not support the installation of a third-party operating system.

Even though Linux is not pre-installed on the PS3, Sony has promoted Linux on the

PlayStation since the release of *PS2 Linux* [41]. Linux has been supporting PS3 since kernel 2.6.21, and Sony even worked in cooperation with Terra Soft developing Yellow Dog Linux 5 for the PS3. However, due to the restrictions of access to the RSX GPU by the hypervisor, accelerated graphics is not possible.

IBM has developed a software development kit for Linux, which provides the libraries, tools and resources needed to develop applications on the Cell BE under Linux[1]. The most basic tools in this SDK are *gcc*, *spu-gcc* and *embedspu*. *gcc* is used for building binaries for the PPE, while *spu-gcc* builds binaries for the SPEs. *ppu-embedspu* converts SPE programs into object files that can be linked into a PPE executable. The Cell BE is also supported by the IBM XL C/C++ compiler [42], which is a cross-compiler for x86 platforms only.

2.4 Parallel programming and programming models

Traditionally, programs have been written with a single thread of control in mind, as most systems have been limited by a single core. Today, multi-core architectures are becoming increasingly more common, which also increases the demand for understanding parallel programming. This style of programming uses multiple processing elements to simultaneously solve a problem, by breaking the problem up in independent parts and processing them separately. Parallel programming introduces many new challenges, as well as strategies and programming models for solving a given problem.

2.4.1 Terminology

Asynchronous: In an asynchronous programming model, different processor elements execute different tasks without needing to synchronize with other processor elements or tasks.

Synchronous: Coordinated in time among tasks, such as when one task notifies a second task about some state change, which the second task receives when it is polling for such notification.

[1]Now part of the non-free IBM SDK for Multicore Acceleration

Synchronization: Synchronization enforces constraints on the ordering of events occurring in different processor elements.

Task: A unit of execution.

Thread: A fundamental unit of execution in many parallel programming systems. One process can contain an arbitrary number of threads, limited by the library used.

Atomic: An atomic operation is uninterruptible. In parallel programming, this can mean an operation (or set of instructions) that has been protected by synchronization methods.

Bandwidth: The number of bytes per second that can be moved across a network. A program is bandwidth-limited when it generates more data-transfer requests than can be accommodated by the network.

Busy wait: Using a timed loop to create a delay. This is often used to wait until some condition is satisfied. Because the device is executing instructions in the loop, no other work can be done until the loop ends.

Critical section: A critical section is a piece of code that can only be executed by one task at a time. It typically eliminates in fixed time, and another task only has to wait a fixed time to enter it. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use. A semaphore is often used for this.

Deadlock: A deadlock occurs when two or more tasks or processor elements are stalled, waiting for each other to perform some action such as release a shared resource.

Latency: Latency is a time delay between the moment something is initiated, and the moment one of its effects begin. In parallel programming, this often refers to the time between the moment a message is sent and the moment it is received. Programs that generate large numbers of small messages are latency-bound.

Load balance: Distributing work among processor elements so that each processor element has roughly the same amount of work.

Monitor: A software monitor consists of a set of procedures that allow interaction with a shared resource, a mutual-exclusion lock, the variables associated with the shared resource, and a monitor invariant that defines the conditions needed to avoid race conditions. A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition. If every procedure guarantees that the invariant is true before it releases the lock, then no task can ever find the resource in a state that might lead to a race condition.

Mutual exclusion: Mutual exclusion (often abbreviated to *mutex*) algorithms are used in parallel programming to avoid the concurrent use of non-sharable resources by pieces of code called critical sections. When several processor elements share memory, an indivisible test-and-set of a flag is used in a tight loop to wait until the other processor elements clears the flag. This test-and-set ensures that when the code enters the critical region, the flag is set. When the code leaves the critical region, it clears the flag. In a spin lock or busy wait mutex, the wait loop terminates when the test finds that the flag is not set, and the wait continues if the flag is set.

Race condition: A race condition occurs when the output exhibits a dependence (typically unexpected) on the relative timing of events. The term originates with the idea of two signals racing each other to influence the output first. The term race condition also refers to an error condition where the output of a program changes as the scheduling of (multiple) processor elements varies.

Semaphore: A semaphore is a protected variable (or abstract data type) and constitutes the classic method for restricting access to shared resources (for example shared memory), in a parallel programming environment. There are two operations on a semaphore; V, often called *up*, and P, often called *down*. The V operation increases the value of the semaphore by one, while the P operations decreases the value of the semaphore by one. The V and P operations must be atomic operations.

Starvation: Starvation occurs when a task tries to access some resource but is never granted access to that resource.

2.4.2 Amdahl's Law and Gustafson's Law

Optimally, one would want the speedup from parallelization to be linear — doubling the number of processing elements should cut the runtime in half. However, achieving this optimal speed-up with a parallel algorithm is very hard to do, and is only plausible when the number of processing elements is low, and most of the program is parallelizable. Speed-up may be near-linear for a small number of processors, until the number of processing elements increases and the speed-up stagnates. This trend is illustrated in figure 2.7.

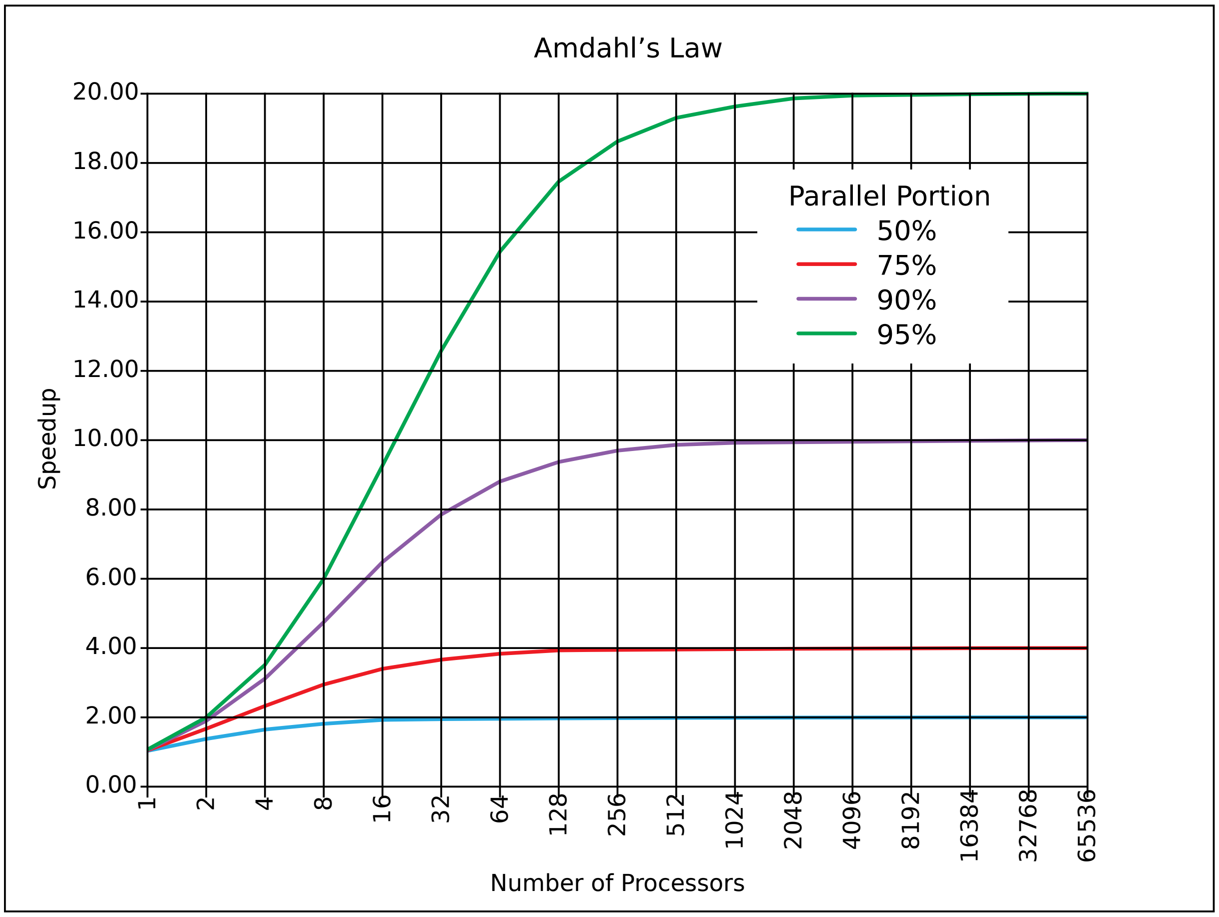

Figure 2.7: The potential speed-up of parallelization according to Amdahl's Law. Illustration is taken from Wikipedia [3].

Amdahl's law [3] is a model for the relationship between the expected speed-up of a parallelized algorithm relative to the serial algorithm, given that the problem size remains the same after parallelization. Any mathematical or engineering problem consists of several parallelizable parts, but also several sequential parts, thus making the potential speed-up limited to the parts parallelizable. This relationship is given by the equation

$$S = \frac{1}{1-P}$$

where S is the speed-up of the program as a factor of its original sequential runtime, and P is the fraction of the program that is parallelizable. Given that only half of the program's runtime is parallelizable, the maximum achievable speed-up is 2, regardless of the number of processing elements.

Closely related to Amdahl's law is Gustafson's law [43], which addresses the shortcomings of Amdahl's law, primarily that Amdahl did not take into account the availability of computing power as the number of machines increases. Amdahl's law assumes a fixed problem size and that the size of the sequential section is independent of the number of processors, whereas Gustafson's law does not make these assumptions. Gustafson's law can be formulated as

$$S(P) = P - \alpha(P-1)$$

where P is the number of processors, S is the speed-up, and α the non-parallelizable part of the program. Gustafson's law implies that, if more processing elements are available, larger problem sizes can be solved in the same time.

2.4.3 Parallelism Challenges

One of the most vital steps to consider when parallelizing a program, is locating and exploiting concurrency. One needs to understand the data flow, data dependencies and functional dependencies of the program to make an efficient parallel implementation, which does not succumb to overhead from data dependencies, synchronization or data-sharing. When looking for concurrency, key elements to examine are function calls, loops and large data structures operated on in chunks. It might be worthwhile to do an analysis of the program, and finding out which parts are the most computationally intensive, using profiling tools like Cachegrind, Callgrind and KCachegrind [2].

Parallelizing a program may lead to a lot of potential overhead, and it is up to the

[2]Cachegrind and Callgrind are part of the Valgrind Tool Suite, and may be found at *http://valgrind.org/info/tools.html*. KCachegrind is a graphical profile visualization tool, and may be found at *http://kcachegrind.sourceforge.net*

programmer to make sure the overhead justifies the improvement in performance. After breaking down the program into tasks that can execute in parallel, one needs to consider data dependencies, task size and grouping, overhead from data transfers in terms of latency, bandwidth and amount of data, and synchronization overhead. In situations where a single task may be too small to justify assigning it to a single processing element, it might be beneficial to group tasks, thus performing several sets of operations on data. This approach reduces the amount of synchronization needed between groups, and may also be worthwhile when tasks share data dependencies.

After choosing a parallelization strategy that suits the program, performing profiling and doing hot-spot analysis may give the programmer insight to critical code sections that might be worthwhile to additionally fine-tune. Fine-tuning is often heavily influenced by the architecture the program is targeted for, as different architectures may handle certain operations, branching and memory access differently. Following are some considerations to include when fine-tuning a program for the Cell BE:

Branching: The SPU in the Cell BE assumes sequential instruction flow, and a branch instruction has the potential of breaking this assumed flow. Branches predicted correctly executes in one cycle, while mispredicted branches (conditional or unconditional) causes a penalty of 18 to 19 cycles, depending on the address of the branch target [44]. Branch instructions also reduce the compilers ability to schedule instructions, by creating a barrier on instruction reordering. As the SPU has a typical instruction latency of two to seven cycles, mispredicted branches may seriously degrade the program performance. The three most efficient methods of eliminating branch prediction are function inlining, loop unrolling, and branch predication.

Function inlining and loop unrolling: Function inlining eliminates the two branches associated with function-call linkage — *branch and set link* (brasl) and *branch indirect* (bi) — for function-call entry and function-call return, respectively. Loop unrolling eliminates branches by reducing the number of iterations, and may be either manually unrolled or automated by the compiler. However, due to the limited space of the SPE's LS, one should be careful with over-excessive use of the latter technique, as it may greatly increase program size.

Predication: It is also possible to eliminate branches for control-flow statements, for example *if* and *if-then-else* constructs. By computing the results of both the *then* and *else* clauses, then using *select bits* (selb) to choose the result as a function of the conditional, the *if-then-else* statement can be made branchless. Additional cycles can be saved if the computation of both results is less than a mispredicted branch [44].

2.4.4 Parallel programming models and strategies

Parallel programming models exist as an abstraction above hardware and memory architectures, as an idealized view onto which applications can be mapped to express concurrency. As examples, parallel programming models may include shared memory, message passing, threads, data-parallelism, and hybrids of these. Although each programming model is not architecture-specific, certain models may be more suitable for a given architecture due to hardware characteristics.

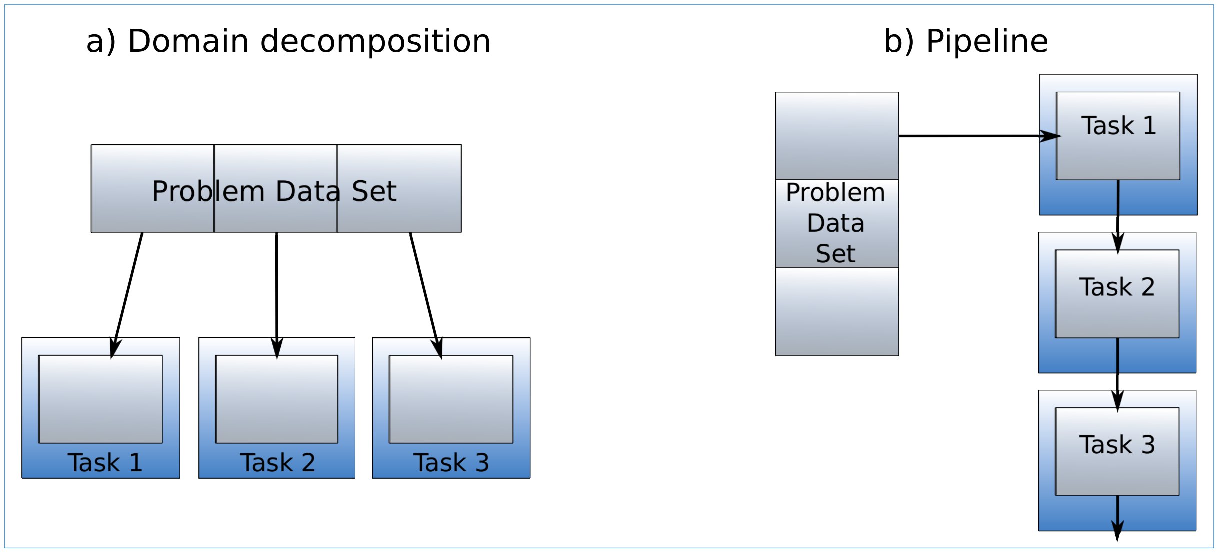

Figure 2.8: Work distribution models.

Work distribution strategies define how concurrency is exploited in an application, and how the program is partioned into independent tasks or grouped into execution threads. Some work distribution models include shared memory, domain decomposition, and pipelining. A shared memory model on the Cell BE usually involves having larger standalone SPE programs, accessing shared memory objects in effective-address space through a locking mechanism. This can be done through an object format called CBE Embedded SPE Object Format (CESOF) [45].

More common is a domain decomposition model, where data is decomposed into sub domains and assigned to a single task, as illustrated in figure 2.8a. This model often utilizes a shared job queue, where each processor elements must acquire a lock to fetch the next job. For uneven jobs, this model has the advantage of self-balancing the load between the processing elements. Where data needs to undergo several stages of computation, e.g. in video coding, a pipeline model may be more applicable. In this model, each stage is represented as a separate task, and data is transferred between them as stages complete, as illustrated in figure 2.8b. Pipeline models have the advantage of utilizing LS to LS DMA bandwidth; However, it is harder to load-balance.

On the Cell BE, work distribution models are often considered either *PPE-centric* or *SPE-centric*. In PPE-centric models, the PPE is responsible for distributing work amongst the SPEs, as well as the loading and switching of SPE programs. In SPE-centric models, application code is usually distributed amongst the SPEs, and the SPEs are responsible for fetching the tasks from main memory. Although a PPE-centric approach may provide finer control over the SPE programs, the PPE load may become heavily utilized by management and control operations, tying it up from processing other tasks.

2.5 Related Work

2.5.1 The MARS framework

The Multicore Application Runtime System (MARS) [4] is a framework for the Cell BE that tries to address the performance overhead of cooperative multitasking on the Cell BE's many SPEs. MARS provides an efficient runtime environment for SPE centric application programs, and a set of libraries that provide an API to manage and create programs. MARS takes advantage of a more lightweight context switch, by letting a microkernel run on each of the available SPEs and take part of the scheduling. This also minimizes the runtime load of the host processor (PPE). MARS is developed by Sony, and released under the MIT license.

When programming on a multi-core architecture such as the Cell BE, there are several caveats one should be aware of. First of all, the local store of the SPE is limited (256KB on the PS3). As the application grows more complex, it is also important for the user to make sure the partioned segments of code do not exceed the physical limits of the local store. Second, the number of physical SPEs is limited (6 on the PS3). As the number of workloads grow, and exceeds the number of physical SPEs, the workloads need to be scheduled. The scheduler also needs to take into account the execution order of the programs, since the programs may interact with each other.

In the rest of this section, we use MARS terminology as in the MARS documentation, where *host* is the host processor (PPE), *MPU* for micro-processing unit (SPE), *host storage* for shared memory space (main memory), *MPU storage* for local memory space (local store), *MPU program* for SPE program, and *workload* as a generic unit of work scheduled for execution on the MPU (SPE context).

The Host- and MPU centric programming model

When using a host centric programming model, the host processor is responsible for organizing and supervising the programs to be run on the MPU. This includes loading and switching programs, as well as sending and receiving necessary data (see figure 2.9). This substantially increases the load on the host processor, and also results in decreased performance of the MPUs, as they often have to wait for instructions and data from the host which causes idle time. There are of course several strategies for optimizing this, however, if the MPUs can directly send and receive data, and load and switch programs, the MPUs might be able to decrease idle time and be used more efficiently.

In the MPU centric programming model, the individual MPUs are responsible for loading, executing, and switching MPU programs. They are also responsible for sending and receiving the data between MPUs (see figure 2.10). These operations are done independently of the host program, making the MPUs self managing. The host is however still responsible for some of the setup management necessary to execute the MPU programs. The MARS framework provides the MPU centric model.

The MARS kernel and workloads

The MARS kernel is a small piece of code that gets loaded in to each of the MPU's storage, and stays resident there during execution. The kernel controls program execution, and is responsible for scheduling the workloads. Based on the workload, the kernel loads and executes the necessary MPU program. The scheduler is non-preemptive,

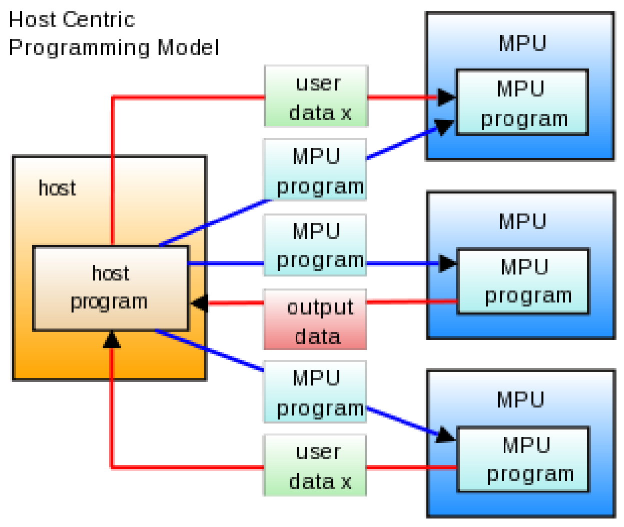

Figure 2.9: A host centric programming model. Figure is taken from the MARS documentation. [4]

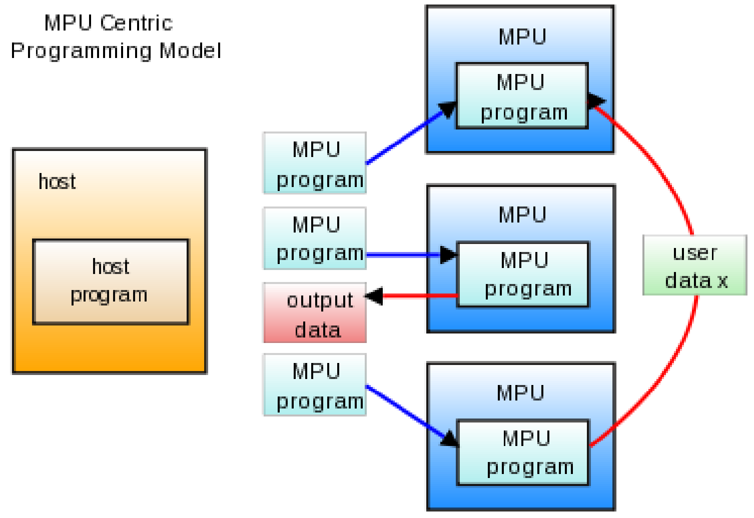

Figure 2.10: An MPU centric programming model. Figure is taken from the MARS documentation. [4]

which means each workload runs until it either finishes, or enters a waiting or blocking state. When this occurs, the kernel performs the context switch.

The kernel has three basic states of operation; schedule, load/execute and save/context switch. Once the kernel has been started, it searches the workload queue, located in host storage, for a ready workload. Every time a workload is created, it is put into the workload queue. Since the queue can be accessed by both the host and MPUs, it is protected by atomic operations. If the scheduler finds an available and ready workload, the workload context is loaded into MPU storage and then executed. If the workload enters a waiting/blocking state, the workload context is saved in host storage, and another workload is loaded if ready. See figure 2.11 for an illustration.

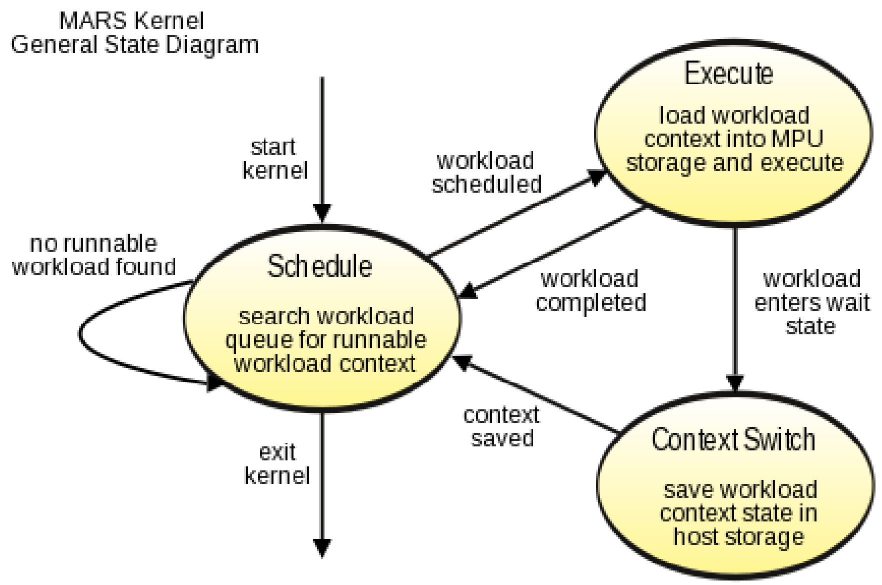
MARS Kernel
General State Diagram
start kernel
workload scheduled
Execute
load workload context into MPU storage and execute
no runnable workload found
Schedule
search workload queue for runnable workload context
workload completed
workload enters wait state
context saved
exit kernel
Context Switch
save workload context state in host storage

Figure 2.11: MARS kernel state diagram. Figure is taken from the MARS documentation. [4]

As mentioned, whenever a new workload is created, it is put into a workload queue. This queue is located inside a MARS context, which must be created at the start of any given program, and destroyed before exit. A MARS context also contains all necessary information and data about the instance initialized. When a context is created, it loads the MARS kernels on to the MPUs, where they reside until the context is destroyed. It is key to only have one MARS context initialized per system, to take advantage of the lightweight context switch MARS offers. More than one MARS context initialized at a time results in dramatic decrease in performance, as the whole context is switched in and out.

2.5.2 OpenCL

The Open Computing Language (OpenCL) [46] is a framework developed by the Khronos Group, for building parallel applications portable across multiple heterogeneous platforms, including multi-core machines, GPUs, and the Cell BE. It has a strong heritage from the CUDA [25] environment, but provides a common hardware abstraction layer across multiple architectures. OpenCL consists of a programming language, called OpenCL C, and a set of OpenCL runtime API functions. The OpenCL C language is a subset of C99, with the addition of support for vector types, images, and memory hierarchy qualifiers. While OpenCL has many publicly available implementations, including releases from Apple [47], AMD [48] and NVIDIA [49], the only implementation fully supporting the Cell BE is part of IBM's XL C/C++ compiler [42,50], which is non-free. There exists another project called OpenCL PS3 [51], which aims to make an OpenCL implementation for the PS3, but this project currently only supports a minimal subset of the specification. In this section, we briefly introduce OpenCL as specified in [46].

OpenCL platform

The OpenCL platforms consists of a host, connected to one more OpenCL devices. These devices are further divided into one or more compute unites (CUs), and a CU consists of one or more processing elements (PEs). An OpenCL application runs on the host, and submits commands to execute computations on the PEs within a OpenCL device. There are three types of commands: kernel execution, memory, and synchronization. Kernels are functions that execute on the PEs, and are further split into two categories: OpenCL kernels, which are written in the OpenCL C programming language, or native kernels, which are accessed through a host function pointer. Both types of kernels are submitted to a device's command-queue in the form of a kernel execution command, and executed on the PEs within that device. Each device may have one or more command-queues, and commands are executed either in-order, or out-of-order, depending on the queue type.

Execution model

When a kernel is submitted for execution, the host defines an index space where instances of the kernel are executed. Each kernel instance is called a work-item, and is identified by its point in the index space. Each work-item is able to query its own ID, and can perform different tasks and operate on different data, based on the its ID. Work-items are further organized in work-groups, which provide a more coarse-grained decomposition of the index space. Each work-group has a unique ID, and each work-item inside the work-group is assigned a unique local ID for that group, making it possible to identify the work-item by either its global ID, or by a combination of the work-group and local ID. Work-items in a work-group execute concurrently on the PEs of a device.

Programming models

While the driving programming model of OpenCL is data-parallel, the platform also supports task-parallel programming models, and hybrids of these. The data-parallel programming model defines computation as a sequence of data elements fed into a work-group, where the index space associated with the execution model defines how the data maps to the work-units. In a strictly data-parallel model, the data elements and work-units map with a one-to-one ratio. OpenCL implements a relaxed data-parallel model, where a one-to-one mapping is not required. Moreover, the programmer may explicitly define how work-items are divided into work-groups, or let the OpenCL implementation implicitly handle the division.

In the task-parallel programming model, a work-group contains only a single instance of a kernel, where the work-group is executed on a CU independently of all other work-groups. Parallelism in this model is expressed by enqueuing multiple tasks which run in parallel within a CU, and by using vector data types implemented by the device. While the data-parallel model is designed for devices such as GPUs, which may have hundreds of cores, the task-parallel model is intended for multi-core CPUs and the Cell BE, where cores are rather limited in comparison.

Memory model

OpenCL defines four distinct memory regions that kernels in a device have access to: global, constant, local and private. Global memory permits read and write to all work-items in all work-groups. Constant memory is allocated and initialized by the host, and remains constant during the execution of a kernel. Local memory is a region local to a work-group, and is used for allocating variables that are shared by all work-items in a group. Private memory is exclusive to a work-item, and is not visible to other work-items.

Synchronization

Synchronization may be performed for work-items in a single work-group, and between commands enqueued to command-queues in a single context. For work-items, synchronization is done by the use of a work-group barrier, which must be executed by all work-items of a work-group, or by none at all. Between work-groups, there is no synchronization mechanisms in OpenCL. For synchronization between commands, command-queue barriers and events are used. Command-queue barriers ensures that all previously queued commands have finished execution inside a single command-queue, while events synchronizes commands in different command-queues. Each API function that enqueues a command returns an event, identifying the command and the memory object it updates. A subsequent command waiting on that event is guaranteed to have an updated memory object before execution.

2.5.3 MapReduce

MapReduce, introduced by Dean and Ghemawat at Google, is a programming model and an associated implementation for processing large data sets on clusters of commodity machines. It is inspired by the map and reduce primitives found in Lisp, and many other functional languages. The framework consists of two user defined functions, *Map* and *Reduce*, and an implementation of a MapReduce library, that automatically handles the scheduling of program execution across machines, input data partioning, machine failure, and the inter-process communication required.

Semantics

In MapReduce, the programmer expresses the algorithm by the two user-defined functions, *map* and *reduce*. The *map* function takes a set of key/value pairs (record) as input, and outputs a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key, and passes them to the *reduce* function. The *reduce* function takes an intermediate key, and a set of values for that key. It will then merge the values to form a possible smaller set of values. Typically, just zero or one output value is produced per invocation to the function.

Listing 2.1 shows the basic structure of a program in pseudo-code, counting the number of occurrences of a word in a collection of documents [13]. The *map* function emits each word in the documents with an associated temporary count of occurrences. The *reduce* sums together all counts emitted for each unique word.

```
// key: document name, value: document contents
map(String key, String value):
  for each word w in value:
    EmitIntermediate(w, 1);

// key: a word, values: a list of counts
reduce(String key, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(result);
```

Listing 2.1: MapReduce pseudo-code

Implementations

The original implementation by Google was written in 2003, and is targeted for processing multiple terabytes of data on very large clusters of commodity machines. Since network bandwidth is a relatively scarce resource in clusters of such dimensions, input data is distributed using the Google File System (GFS) [52], and stored on the local disks of the machines that make up the cluster. GFS also stores replicas of data on multiple nodes to improve reliability and fault tolerance.

Hadoop [53] is an open-source MapReduce implementation, developed by Apache. Hadoop distributes data across networked machines using the Hadoop Distributed File System (HDFS), which also creates multiple replicas of data blocks for better reliability. A HDFS cluster primarily consists of a *NameNode*, and several *DataNodes*. The *NameNode* runs on a single master machine, managing file system meta-data and information about the other machines in the cluster, while the *DataNodes* store the actual data. Above the file system is the MapReduce engine of Hadoop, which consists of a single *JobTracker*, and multiple *TaskTracker* nodes. The *JobTracker* usually runs on the same master machine as the *NameNode*, and pushes out jobs to the *TaskTracker* nodes in the network as they become available.

Phoenix [20] is an implementation of MapReduce targeted for shared-memory multi-processor systems, with an associated runtime time that handles resource management and fault tolerance. The runtime uses POSIX threads to spawn parallel Map and Reduce instances, and schedules tasks dynamically to available processors. In addition to the Map and Reduce functions, the user provides functions that partition the data before each step, and a function that implements key comparison [20].

Mars[3] [21] implements the MapReduce framework for graphics processors. It assumes a flat view of the GPU memory, and relies on the GPU's heavy multi-threading to hide latency and sustain high bandwidth. Since GPUs do not support dynamic memory allocation, device memory for input data and result data is allocated before executing the program. Sorting is accomplished by using a high-performance bitonic sort tailored to the underlying architecture [21]. Unlike Google's MapReduce and Phoenix, Mars does not partition map output data before sorting.

Kruijf and Sankaralingam [22] have developed an implementation of MapReduce for the Cell BE, with the intent of hiding the underlying complexity of the architecture. Due to the explicit memory transfers required between SPEs and main memory, the implementation pre-allocates map and reduce output regions for bulk DMA transfers. The implementation also parallelizes computation with memory transfers by double buffering and streaming data whenever possible [22]. However, the implementation does not support distributed environments, and is limited to a single machine.

Alternatives to MapReduce

Although MapReduce has become one of the more recognized frameworks for developing distributed and parallel applications, there are also many alternatives. The Pig Latin programming language, developed by Yahoo, is designed to fit between the declarative style of SQL, and the rigid paradigm of MapReduce [16]. Pig Latin programs are compiled into map-reduce jobs, which is then executed using Hadoop. DryadLINQ [54] is a programming environment developed by Microsoft, consisting of the Dryad distributed execution engine, and the .NET Language Integrated Query (LINQ). Unlike MapReduce's data-parallel model, Dryad uses a task-parallel model

[3]Must not be confused with the framework presented in section 2.5.1

to express computations in a directed acyclic graph, where the vertices in the graph are executed on a set of clustered machines. Dryad also has the ability to manage dependencies between vertices, and recreation of vertices in case of machine failures. As the relation between the distributed file systems and the execution engines are considered relatively loosely coupled in many systems, Cogset [55] addresses this by fusing the two components to a single platform, allowing native support for parallel processing on the actual storage nodes. Cogset also increases performance by changing the software architecture, reducing overhead from higher level programming abstractions layered on top of MapReduce, like Pig Latin and Sawzall [56]. Finally, supporting process structures containing branches and cycles is the goal of Nornir, a runtime system for executing KPNs, which is a shared-nothing, message-passing model of concurrency.

2.6 Summary

In this chapter, we have presented an overview of multi-core architectures, and the Cell Broadband Engine in detail. We have also presented the technical challenges that comes with multi-core architectures, and the techniques used to tackle them.

MARS offers a promising programming model for the Cell BE, particularly with the implementation of a microkernel, allowing lightweight context switches on the SPEs. However, the current versions of MARS have no support for dynamically loading programs to the SPEs, and requires them to be embedded with the main application. This makes the framework unsuitable for distributed environments, where the offloaded routines are sent to nodes during runtime. With regard to OpenCL, there is currently only one compiler that has support for the Cell BE, and that is IBM's XL C/C++ compiler. However, since this compiler is non-free, we were not able to test it.

MapReduce is a general programming model, inspired by the map and reduce primitives of functional languages. The main implementation by Google in 2003 was targeted for processing vast amounts of data, in large clusters consisting of commodity machines. In the later years, several architecture specific implementations has been made, to ease the programming effort for multi-core and heterogeneous platforms, including GPUs and the Cell BE. Alternatives to MapReduce have also been introduced, each with their strengths and weaknesses. However, while frameworks exist for individual platforms, there is currently no framework that is jointly able to exploit the processing potential of multiple heterogeneous elements in a distributing environment. Moreover, the MapReduce model is not general enough to cover all application domains. Ranger et al. [20] performed an evaluation of MapReduce on CMP and SMP platforms, and compared performance with parallel code written directly in POSIX threads. They achieved similar, or slightly better performance for applications that fit naturally into the MapReduce model, like word count, string match, and matrix multiplication. However, for algorithms that is not an efficient fit, like k-means, principal component analysis (PCB), and histogram, POSIX threads outperformed MapReduce significantly. In addition, the key/value paradigm requires major re-writes of existing applications.

Due to the reasons stated above, we have decided to investigate the design of a general purpose framework for offloading compute intensive task to heterogeneous elements in a distributed environment. In the following chapters, we discuss the design challenges implicated with such a framework, and implement a prototype for offloading tasks to the Cell BE. The prototype will be evaluated by adapting applications, and by benchmarking scalability of added hardware. In chapter 3, we discuss design challenges, and propose a a design for the framework. In chapter 4, we realize the design, and implement the framework prototype, consisting of a runtime library, and a scheduler for the Cell BE. In chapter 5, we use the framework to offload the DCT and quantization steps of a Motion JPEG video encoder, and in chapter 6, we offload data encryption with the Advanced Encryption Standard block cipher. In chapter 7 we discuss results and lessons learned, and in chapter 8 we summarize and conclude our thesis, and propose further work.

Chapter 3

Design

In this chapter, we present the design of our prototype framework, required for the implementation of the runtime library and Cell BE scheduler. As described in section 2.6, the MapReduce paradigm is not general enough to be applicable to all application domains. Therefore, we want to investigate a more general purpose framework, where compute intensive tasks can be offloaded to a heterogeneous distributed environment.

We start this chapter by giving a general introduction to the ideas behind the framework in section 3.1. We then move on to presenting the design overview for the framework in section 3.2. In section 3.3, we discuss the design details of the runtime library, and in section 3.4 we discuss the details regarding the Cell BE scheduler. The design is summarized in section 3.5.

3.1 Introduction

With the proliferation of heterogeneous platforms and high-performance networked devices, vast amounts of potential processing power has become available to programmers and end-users. However, harnessing this potential is a challenging task. The complexity of programming for each architecture varies between platform, as some require highly optimized code to make the transition beneficial. Frameworks for heterogeneous architectures tend to be fairly complex, and requires specific parallelization strategies to perform well.

To hide the underlying complexity of each architecture, we propose a framework prototype that is able to harness the potential processing power of multiple platforms, in a heterogeneous distributed environment. In this chapter, and further in the thesis, we define the *client* as the user application, *nodes* as the networked Cell BEs, and the *scheduler* as the scheduler daemon running on the Cell BEs.

3.1.1 Scenario

The user writes custom routines for a target architecture, and specify these in a *workload*. The workload is also assigned the partitioned data, which is processed by the routine specified. As part of the framework, a runtime library is linked with the application. The runtime is able to manage workload distribution across the applicable nodes, and handle synchronization events. The workload is then dispatched to a node, processed remotely, and results are returned to the client. Each of the nodes in the cluster runs a workload scheduler, optimized for the target architecture. The scheduler is able to queue incoming workloads from multiple clients, and process the data as specified by the routine in the workload. The scenario is illustrated in figure 3.1.

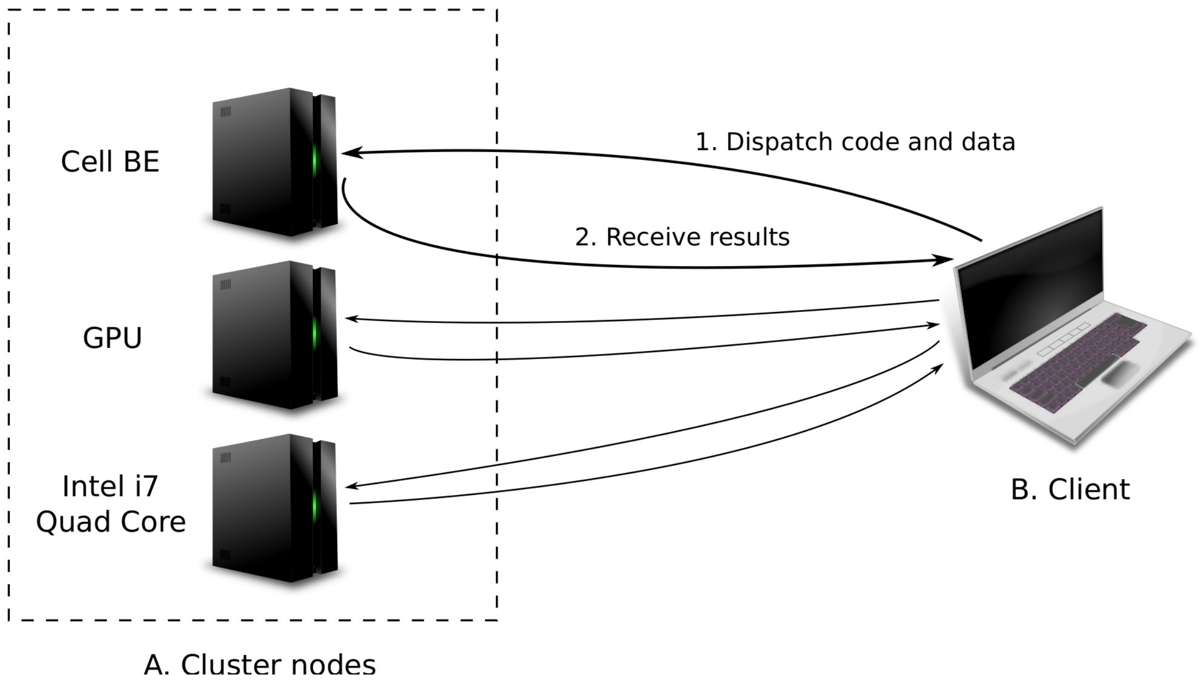

Figure 3.1: Distributed offloading scenario.

3.1.2 Concept and requirements

The intent with this framework, is to let the programmer define workloads, which are transparently distributed across nodes in a network by the runtime library. Each of the nodes run an architecture specific scheduler, which processes the workloads, and sends the output data back to the client. A framework like this poses several challenges, but also many opportunities.

The workloads defined by the client contains the data to be processed, the target platform, and the routine for processing it. Much like the user defined map and reduce functions in MapReduce, the user writes an architecture specific routine for the workload, allowing exploitation of data-level parallelism, like SIMD for Cell BE's SPEs, and Intel's SSE instruction set. Contrary to MapReduce, the workload approach also allows for task-parallelism, as the programmer may define different functions in each workload. The workloads are dispatched by a runtime library in the framework, linked with the application at hand. The runtime is able to transparently handle workload distribution across the target nodes, load balancing, and synchronization events.

One of the key challenges with the framework, is the architecture specific scheduler. The scheduler runs on each node in the network, and processes workloads dispatched by the clients. For x86/x64 multi-core architectures, the scheduler is relatively straight forward, in that workloads can be assigned to a thread and executed on separate cores. However, for GPUs and the Cell BE, the scheduler is more complex. The hardware architecture of GPUs differs significantly, in that GPUs provides finer grain thread-level parallelism on hundreds of streaming processors (SPs), which needs to be taken into account when constructing the workload. Furthermore, GPUs do not support dynamic memory allocation [21], which further complicates the scheduler. With regard to the Cell BE, challenges arise due to the three-level memory model, and the explicit management of DMA transfers. Program execution on the SPEs also needs to be handled efficiently, due to high context switch overhead [57].

These are just some of the challenges that needs to be taken into account when designing such a framework. While we would like to further investigate the integration of multiple platforms, we have due to time constraints limited the scope of this thesis to the Cell BE, and leave the remaining platforms as further work. In the following sections, we present the design of a prototype framework, including a runtime library, and a scheduler for the Cell BE.

3.2 Design overview

The framework consists of two main components, as illustrated in figure 3.2 — the runtime library, and the Cell BE scheduler. The Cell BE scheduler acts upon requests from the runtime library, assigning processing elements to dispatched workloads. In this context, a workload consists of data, and the corresponding function for that data. When the client wants to offload compute intensive parts of the program to the nodes, it checks if the target contains the code to run the task at hand. If the node does not possess the code, the client transfers the code prior to the workload. The client then registers locally that the node possesses the code, to avoid further network overhead.

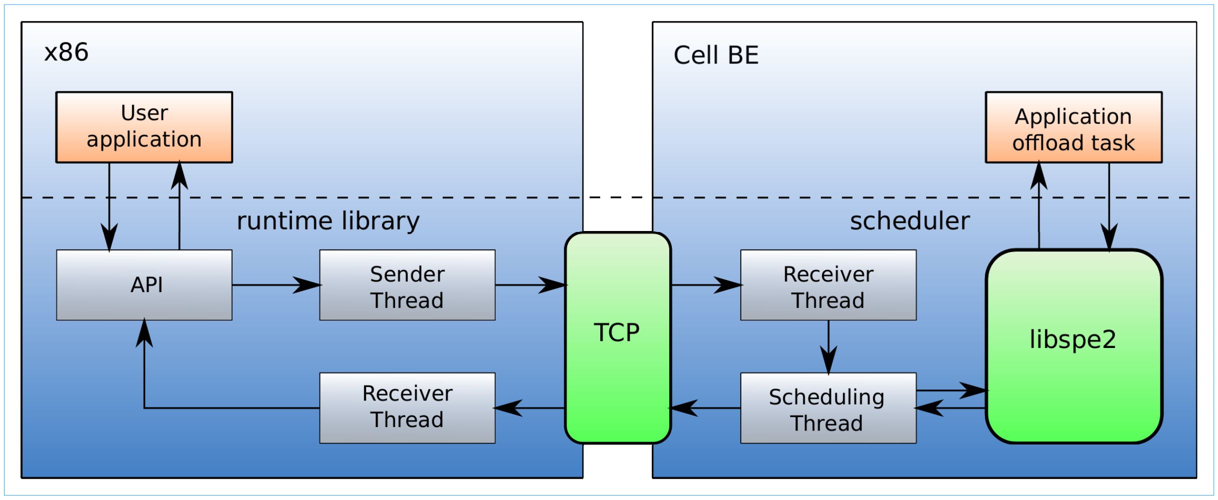

Figure 3.2: System flow.

Should the target machine possess the code from an earlier request, the workload is transferred and queued, then scheduled for execution. When execution is complete, the output data is transferred back to the client, where it is synchronized with the other concurrent workloads. The Cell BE scheduler supports multiple clients, maintaining a unique connection for each one.

During the setup routine, each client program specifies the number of network nodes it wants to use, then specifies the host address of the target machines. While the nodes may be geographically dispersed, network latency and bandwidth will affect the performance of the system, so it is advised having high speed access to the nodes.

Further, we describe the design of the runtime library and Cell BE scheduler, respectively.

3.3 Runtime library design

As mentioned in section 3.2, the client dispatches code and workloads to the Cell BEs, offloading compute intensive parts of the program. This is done by utilizing a runtime library developed for use in conjunction with the Cell BE scheduler, presented in section 3.4. In this section, we describe the design choices made for the runtime library, to make inter-platform communication and synchronization as transparent as possible to the client.

3.3.1 The Workload

In our system, a workload is considered a unit of work, occupying a processing element for an arbitrary amount of time. The workload does not contain the code itself, just the name of the offloaded function, as this is pre-sent by the runtime library each time a new type of function is called. This is to avoid additional overhead by sending duplicate SPE programs. Further, the workload may contain the data to be processed, if it is required by the offloaded function.

In addition to input data and function name, the workload also includes a header. The header contains all information needed for the workload to be sent from the client, processed by the assigned target machine, and sent back to the correct client. The header is constructed by the runtime library, and contains fields with sequence numbers, data sizes, output data addresses, and optional arguments which may be set by the client application.

3.3.2 Runtime configuration

The runtime library requires the programmer to set up a small configuration before utilizing system. This involves defining a set of target machines running the scheduler, allowing the runtime to set up data structures and connections. As mentioned in section 3.2, the machines may be geographically dispersed, but this will most likely affect the performance of the application. The client is allowed to define any number of nodes, as the runtime will perform load balancing when distributing the workloads.

The runtime spawns a receiver thread for each unique node defined by the client, which is responsible for managing output data sent by the Cell BEs. As mentioned in section 2.3.4, the Cell BE uses big-endian ordering, so output data needs to be converted accordingly, if required by the data type. To avoid busy-waiting on the socket, the receivers initially sleep on a condition variable, and does not proceed to read on a socket until waken up by a workload transmission. When output data for the respective workload has arrived, the receivers are responsible for writing the data to the output address defined in the workload, and announcing synchronization events to the client.

3.3.3 Workload transmission

Once the runtime has been properly configured with target machines, the client may start to dispatch workloads to the nodes. As mentioned in section 3.3, we want the underlying components of the framework to be transparent to the programmer, thus only requiring a minimum configuration before allowing transmissions. The system flow of workload transmission is illustrated in figure 3.2

Should the SPE program require input data, the programmer needs to consider the restrictions of the DMA engine in the Cell BE when performing data partitioning. As covered in section 2.3.3, DMA transfers requires data being aligned on a 16 byte boundary. The programmer does not need to explicitly align data to 16 bytes, as this is done by the runtime library, but data must be partitioned with a multiple of 1,2,4,8 or 16 bytes. With regard to the SPE program for processing the data, as explained in section 3.2, the client needs to check if the scheduler possesses the code required. Instead of polling the node for each workload request, which would cause excessive overhead, we register each SPE program transfer locally, and perform a client side look-up.

We considered two different design approaches to the function that dispatches the workload; In the first approach, it could be implemented as a blocking function, i.e., it would not return until the workload had been processed remotely. This would imply that, to achieve concurrency in the application, the function would be need to used in conjunction with a thread library. However, in a scenario where we would want to transmit a thousand concurrent workloads, we need to spawn a thousand threads, which could lead to several problems. First, spawning such a high amount of threads is highly limited by system resources. The POSIX standard does not dictate the size of a thread's stack, and sizes varies greatly among architectures. On the clients in our test setup, the default stack size for each thread spawned is 8 MB, thus, exceeding the stack limit is easy to do. Secondly, each thread would need to know when their respective workload has returned, thus requiring synchronization mechanisms with the receiver threads, potentially causing additional overhead.

Although the stack problem in this approach may be solved by reducing each thread's allocated stack size, synchronizing a thousand threads by, e.g., a condition variable might still cause a lot of overhead. Therefore, we have opted for a second approach, where the function returns to the caller, and workloads are synchronized by a separate routine. This routine checks a sequence number window set during transmission, and blocks until each workload's output data has returned from the network. By choosing this approach, we save system resources and avoid excessive overhead associated with thread synchronization.

3.4 Cell BE scheduler design

As mentioned in section 3.2, the scheduler queues and schedules incoming workloads for processing on the Cell BE. In this section, we describe the design choices made for the scheduler, to make it as lightweight and effective as possible. We discuss potential issues and challenges we had to consider with regard to network latency, as well as scheduling policy and code management.

3.4.1 Managing connections

To be able to communicate with the Cell BEs, the runtime library needs to establish a connection with the scheduler. The scheduler must be able to manage and monitor incoming connections from several clients, allocating resources accordingly. The connections also needs to be addressable by other components in the system, allowing transmission of data back to the clients. The connection manager is run by a single thread. In addition to managing connections and receiving data, it is also responsible for queueing incoming workloads.

In our system, the scheduler is not aware of what kind of applications the clients are running, thus having equal criteria for all incoming workloads. Since dropped packets may be critical for the client, it is crucial that the transmissions are reliable and is able to handle packet loss. We can choose to either transmit workloads with the User Datagram Protocol (UDP) [58], or the Transmission Control Protocol (TCP) [59].

UDP is a simple transmission model, without any mechanisms for providing reliability, ordering, or data integrity. Although it is the fastest of the two protocols, it is certainly the least reliant, as it does not guarantee data delivery. TCP however, is a protocol that guarantees data delivery and correct data ordering. Upon detection of a corrupted packet, TCP retransmits the packet. It also discards duplicate packets, and provide congestion control. Although it is the most reliant protocol, it also comes with the most overhead. As mentioned in section 2.4.3, parallel programming is cumbersome in the first place, and we do not want the client to additionally cope with workload retransmission. Therefore, TCP is our protocol of choice, despite the potential overhead.

3.4.2 Workload queue

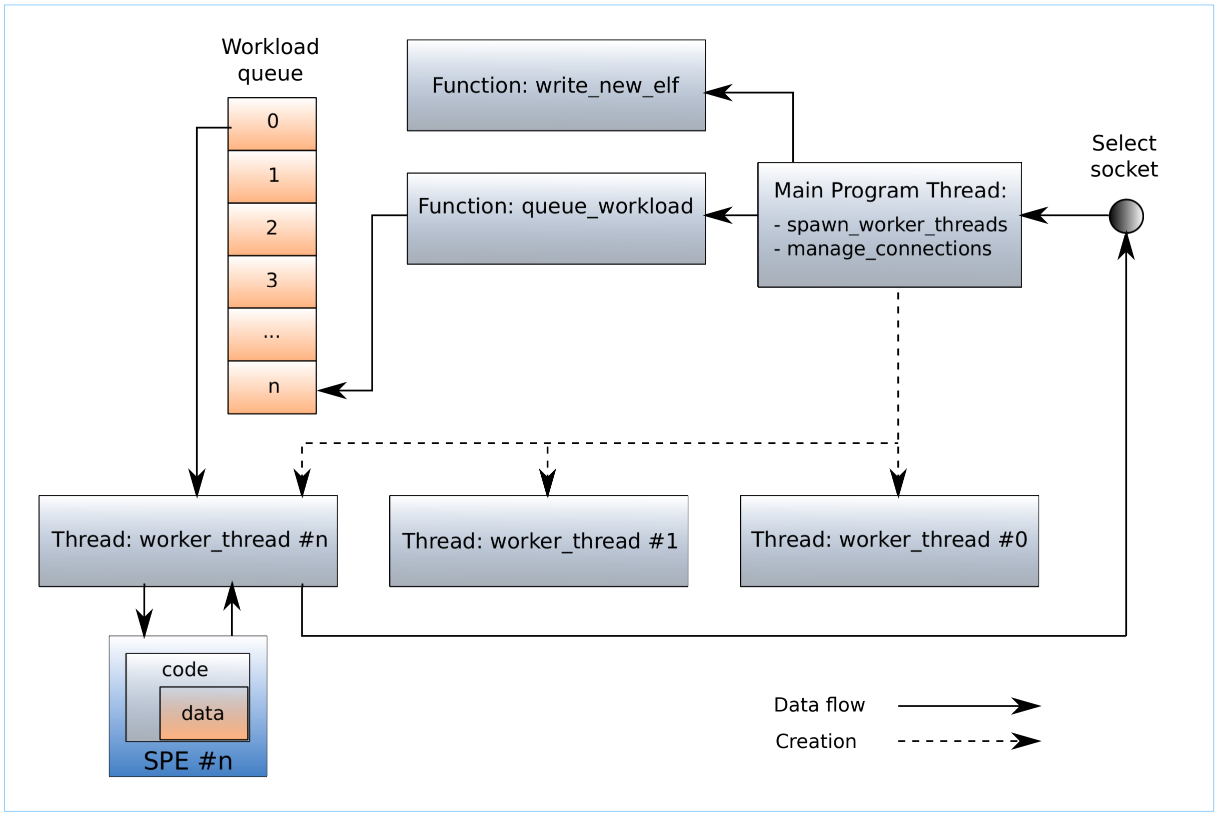

Figure 3.3: An overview of the workload scheduler.

As seen in figure 3.3, workloads are queued directly from the connection manager in the main program thread. The queue is constructed as a first in, first out (FIFO) buffer, which initially is subject to race conditions between queuer and worker threads. Since we have a multi-consumer system (worker threads), the queue needs to be protected by a mutex, effectively avoiding race conditions. Further, to avoid high CPU consumption because of polling the queue (busy waiting), the queue must be wrapped with a condition variable, which allows for the queuer to notify the workers of new workloads. We consider all incoming workloads to have the same priority, thus always inserting the newest workload at the tail of the queue.

3.4.3 Worker threads

The scheduler's setup routine spawns an appropriate number of worker threads. As mentioned in section 2.3.5, the Cell BE in the PS3 has six available SPEs. Thus, to avoid the high context switch overhead, each scheduler creates six SPE contexts, and spawn six worker threads. These worker threads are responsible for popping workloads from the workload queue, scheduling them for execution on the SPEs, and sending results from the computation back to the client.

Initially, after the worker threads have been spawned by the main execution thread, each worker tries to acquire the lock associated with the condition variable for the workload queue, effectively making them sleep on a semaphore until data arrives. Once a workload has been queued, as described in section 3.4.2, the threads are awakened by a broadcast, and attempts to re-acquire the lock. The queue counter has now been incremented, and allows for a worker to pop a workload from the queue.

Scheduling the workload

When acquiring workloads from the queue, we need consider two scenarios with regard to scheduling; Since the system is able to handle multiple unique clients, workloads are likely to differ for each connection. Second, a single unique client may be offloading different parts of a program, causing different workloads to arrive on the same connection. Either way, in a worst case setting, every other workload in the queue will differ. Without an efficient scheduling algorithm, this requires a new SPE program to be loaded to the SPE context for each workload, which in turn causes a lot of overhead. An SPE context is a base data structure required by libspe2, holding persistent information about the logical SPE, and should only be accessed by libspe2 API calls. The calls for destroying and loading SPE programs are expensive in terms of operations, and should be minimized. While we would like to use the MARS framework for scheduling and executing workloads, MARS has no support for dynamically loading SPE programs, as discussed in section 2.6. We therefore consider two different approaches to workload scheduling; A queue-centric algorithm, and an SPE-centric algorithm.

In a queue-centric scheduling algorithm, each worker thread is assigned an SPE context during initialization, and only executes workloads in this context. At first pass, the worker loads the image required by the first workload acquired, and execute the workload at hand. The SPE program loaded is not destroyed, and stays resident in the context. On the next pass, instead of popping the first workload from the queue, the worker searches for a workload requesting the SPE program currently loaded to the context. If no match is found, the first workload in the queue is selected. Depending on the size of the queue, this approach may cause each thread to hold on to the queue lock for a considerable amount of time, effectively preventing other workers from entering the queue, in turn causing SPEs to idle. Holding the queue lock also prevents workloads from entering the queue, further stalling the connection manager.

The second approach, is an SPE-centric algorithm, where the worker threads are not explicitly assigned to an SPE context. In this scheme, the worker threads pop the first workload available from the queue, and search for a matching SPE context. To find the most suitable context, the workers search through the contexts with the following criteria:

1. The context is available, and has a matching image loaded.
2. The context is available, but no image is loaded.
3. The context is available, but has no matching image.

Since we during initialization spawn six worker threads, one for each context, there is always one available context to the worker. This approach greatly reduces the access time to the workload queue, which in turn reduces idle time for the other workers. Reducing access time also allows the connection manager to queue incoming workloads faster. Due to these advantages, we have opted for the SPE-centric approach in our system.

Executing the workload

During the main thread's setup routine, the scheduler creates six SPE contexts, one for each worker thread. These contexts are needed to load the SPE program images sent by the client to the SPE's LS, prior to execution. As explained in section 3.4.3, we choose to search for a suitable context for the workload, preferably one that already has a matching program loaded.

If a matching context for the workload is found, we only need to reset the entry point for the context, which determines where we start execution. It is possible to load several programs to a single context, and switch between them using different entry points. However, since the LS on the Cell BE is very limited in size, we have chosen to let each program take full advantage of the LS. Where workloads only occupy a small amount of the LS, the scheduling algorithm could optimally load several workloads to the LS at a time, and sequentially process each of them by changing the entry point. Due to time constraints, we do not consider this scenario, and leave it as further work.

Further, if no matching context is found, we either need to load the required program to an empty context, or to a context containing another program. Overhead from destroying, creating and switching contexts is costly, and needs to be avoided. By design, programs loaded to the SPE will must run to completion, effectively allowing us to overwrite the program without the need to create a new context for different workloads.

With regard to the explicit memory transfers between main memory and each SPE's LS, the workload header contains addresses for both input data and output data, and transfers are done from the SPE program. The output data size is specified in the workload header, and allocated before execution. Once execution is complete, the output data is sent back through the socket directly from the worker.

3.5 Summary

In this chapter, we have discussed the design challenges with regard to a multi-platform framework, and presented a design for a prototype targeting the Cell BE platform. The main tasks of the runtime library is to dispatch workloads, manage connections, and manage synchronization events for the client. Each of the nodes in the network runs a scheduler, able to queue process incoming workloads from the client.

In the next chapter, we move on to see how the features of the runtime library and Cell BE scheduler are realized, when we present the implementation specifics.

Chapter 4

Implementation

In this chapter, we present the implementation details of our prototype framework. We go into the details of how the runtime library is constructed, and how the API is used by applications. We also cover the details regarding the Cell BE scheduler, and the communication between the scheduler and runtime library. We start this chapter by presenting the implementation details of the runtime library, then move on to the Cell BE scheduler.

4.1 Runtime library

In this section, we present the implementation details of the runtime library. We give a brief description of how an application should use the provided API, and then go into the implementation details surrounding the different components of the runtime library.

4.1.1 API usage

The runtime library provides the programmer with the *workload_t* data type, and the five functions *cellsched_setup_hosts*, *cellsched_add_host*, *cellsched_send*, *cellsched_sync* and *cellsched_destroy*.

We have provided a complete example usage of the library with pseudo-code, in listings 4.1 to 4.3. The number of concurrent workloads that can be created greatly depends on how often the application needs to synchronize, data partitioning, and the computational load of the workload, as discussed in section 2.4.4. However, for the sake of this example, we do not go into the specifics of the SPE program, or the workload itself, as the example provided is meant to show basic usage of the API. We assume an arbitrary input size with 16kb workloads.

In listing 4.1, we initialize data structures and establish connections with the nodes. First, we include the *cellsched.h* header file, which contains the prototyped functions and the *workload_t* data type. Next, we set the number of hosts, and add the target machines. We use two target machines in this example, although there is currently no limit.

```
#include <cellsched.h>
#include <pthread.h>

int main(void) {
  uint8_t *in_data;
  uint8_t *out_data;

  cellsched_setup_hosts(2);
  cellsched_add_host(''cell-machine1.host'');
  cellsched_add_host(''cell-machine2.host'');

  ...
```

Listing 4.1: Setup routine

In listing 4.2, we first specify the number of workloads we want to send to the Cell BEs before synchronizing. Further, we create a workload on lines 7 to 15. The workload needs to be assigned an SPE program, data addresses, and data sizes. The programmer may also assign optional arguments to the workload, which will be accessible from the SPE program. Once the workload has been created, we send the workload by calling *cellsched_send* with the workload as argument, on line 17.

```
1   ...
2
3   int nworkloads = input_size / workload_size;
4   int bufpos = 0;
```

```

  for(i = 0; i < nworkloads; i++) {
    workload_t load;
    load.elf_img = ''spu_prog.elf'';
    load.in_data = in_data + bufpos;
    load.in_data_size = 1024*16;
    load.out_data = out_data + bufpos;
    load.out_data_size = 1024*16;
    load.dtype = DCHAR;
    load.uint32_arg[0] = optional_arg0;
    load.uint32_arg[1] = optional_arg1;

    cellsched_send(load);

    bufpos += 1024*16;
  }

  ...
```

Listing 4.2: Creation and execution of workloads

Finally, we synchronize the workloads in listing 4.3, before bringing down the connections and exiting the application.

```
  ...

  cellsched_sync();

  ...
  cellsched_destroy();
  return 0;
}
```

Listing 4.3: Synchronization and shutdown

The use of the runtime library entails the need to specify *-lcellsched* as a linker option to *gcc*, as done in listing 4.4.

```
$ gcc -lcellsched program.c -o program.elf
```

Listing 4.4: Compiling

In the following sections, we go into the implementation specifics of the different components in the runtime library.

4.1.2 The workload header

The workload header is created by the runtime library, and contains all information needed to process the workload and send the output data back to the client. The header is shown in listing 4.5.

```
typedef struct _workload_t {
  int32_t    pkg_size;
  int32_t    img_check;
  int32_t    socket;
  uint32_t   seqnr;

  uint8_t    elf_img[32];
  uint64_t   in_data;
  uint64_t   out_data;
  int32_t    in_data_size;
  int32_t    out_data_size;
  int32_t    dtype;

  uint32_t   uint32_arg[4];
} workload_t;
```

Listing 4.5: workload header

The first field in the header is *pkg_size*, which is the total size of the workload, including data. *img_check* is a flag set by the client when pre-sending the SPE program image, so that the scheduler may process the packet accordingly. *socket* is the file descriptor set by the scheduler's connection manager, so that the worker threads know where to send the output after executing the workload. *seqnr* is the sequence number of the workload, used for client synchronization. *elf_img* is the name of the SPE program image, needed by the worker threads. *in_data* and *out_data* is the address of the input and output data, needed by the runtime library. *in_data_size* and *out_data_size* is the input and output data size in bytes. *dtype* sets the data type, which can be either 8, 16 or 32 bit. *uint32_arg* provides optional arguments.

4.1.3 Receiver threads

Each call to *cellsched_add_host* spawns a receiver thread, which works as a connection manager for the given host. The receiver thread is responsible for reading and synchronizing incoming output data from the Cell BEs.

The receiver threads are initialized by *cellsched_setup_hosts*, and spawned by *cellsched_add_host* for each connection. Initially, all threads sleep on a condition variable, until woken up by a workload transmission. The condition variable is associated to a packet counter, which is incremented by the sending threads. Once woken up, they attempt to read from their respective socket, until data arrives back from the host. The data type is specified in the packet header, and endianess is converted accordingly. Each workload has an attached sequence number in the header, so when data arrives, the receiver thread knows which workloads has been completed. Since TCP might pack output data from more than one workload at a time, the buffer supplied to *recv* is implemented as a circular buffer, thus avoiding additional memory copies.

4.1.4 Workload transmission

Workloads are dispatched to the Cell BEs with the *cellsched_send* function, which takes the data type *workload_t* as argument. The function first needs to select a socket for the transmission, as we want to distribute the workloads evenly across the nodes in the network. Ideally, we would want a full fledged load balancer, able to poll the load on the nodes, and choose a node accordingly. However, due to time constraints, we assume the same workloads for all nodes, and use a round robin scheme for node selection. A more efficient load balancer is left as further work.

Before sending the workload to the chosen node, we check if the scheduler possesses the SPE program image required. This is a check done client side by a look-up table, as we do not want to poll the scheduler for each workload sent. If the SPE program is not registered in the table, the SPE program image is wrapped in a workload header, and the *img_check* flag is set. This flag tells the scheduler to process the packet as a program image, where it writes the image to the local file system, making it available for loading with libspe2's *spe_image_open* and *spe_program_load*. The scheduler then sends an acknowledgement back, and the image is registered in the look-up table for the corresponding node.

As covered in section 2.3.4, the Cell BE uses big-endian ordering for data storage and instructions. The workload header needs to be constructed with regard to this, and depending on the data type, input data may need to be converted. In addition to the header fields provided by the the client, the runtime library assigns a sequence number, the workload packet size, and a socket number to the workload. Once the workload is sent, a packet counter for the corresponding socket is incremented, and the receiver threads are signaled.

Since we opted to implement *cellsched_send* as a non-blocking function, workloads must be synchronized by some manner. This is done by calling *cellsched_sync*, which blocks until output data for all workloads up to the point of the call has returned from the nodes. The function checks the sending window of all workloads transmitted, and if output data for a workload has not arrived, it yields to another thread by calling *sched_yield*, and checks the window again when re-scheduled for execution.

4.2 Cell BE scheduler

In this section, we present the implementation details of the Cell BE scheduler. We describe the SPE Runtime Management Library version 2, which is used for executing programs on the SPEs, and then go into the implementation details surrounding the different components of the scheduler.

4.2.1 SPE Runtime Management Library version 2

The SPE Runtime Management Library version 2 (libspe2) is used to control SPE program execution and resource management from the PPE, and makes use of the SPU File System (SPUFS) as the Linux Kernel Interface. libspe2 handles the SPEs as virtual objects called SPE contexts, which means from a programming point of view, SPE programs can be loaded and executed by operating on SPE contexts [5].

We could have chosen to write our own API using the intrinsics the Cell BE provides. However, the Cell BE is a fairly complicated processor, and a lot of code is required to keep track of the system, handling interrupts and writing to registers. We might save a few processing cycles and avoid some degree of overhead, but we feel the package libspe2 provides makes up for this in terms of flexibility and error handling. Also, libspe2 has an interface we are familiar with, which makes it easy for us to use.

libspe2 API

The libspe2 library provides an API for the programmer, which is built around the notion of an SPE context. The SPE context is a virtual object, representing the current state of an SPE, including the data and code loaded to it.

The API defines a number of data types, used as arguments to various API functions. The most important of them are as following:

- `spe_context_ptr_t`
- `spe_program_handle_t`
- `spe_stop_info_t`

The first, *spe_context_ptr_t*, provides a virtual state of the SPE, including LS and registers. This is one of the core data types of the API, which the application loads and executes programs with. The second, *spe_program_handle_t*, is a handle that can be used to identify an SPE program, which is either built and embedded into the PPE program, or opened by the program dynamically. Last, *spe_stop_info_t* is a data type used to record the reason an SPE program stopped execution. This data type is very useful for debugging SPE programs.

The API also provides a set of functions, giving our application the flexibility we need. The most important of these functions are given in the following list, and are usually executed in their respective order:

1. `spe_context_create`. Create a new logical SPE context.
2. `spe_image_open`. Open the SPE program image stored in a ELF executable file.
3. `spe_program_load`. Load the SPE program into the LS.
4. `spe_context_run`. Requests execution of the program loaded in the SPE context.
5. `spe_context_destroy`. Destroy the logical SPE context. This happens automatically if the program exits abnormally.
6. `spe_image_close`. Close the SPE program image.

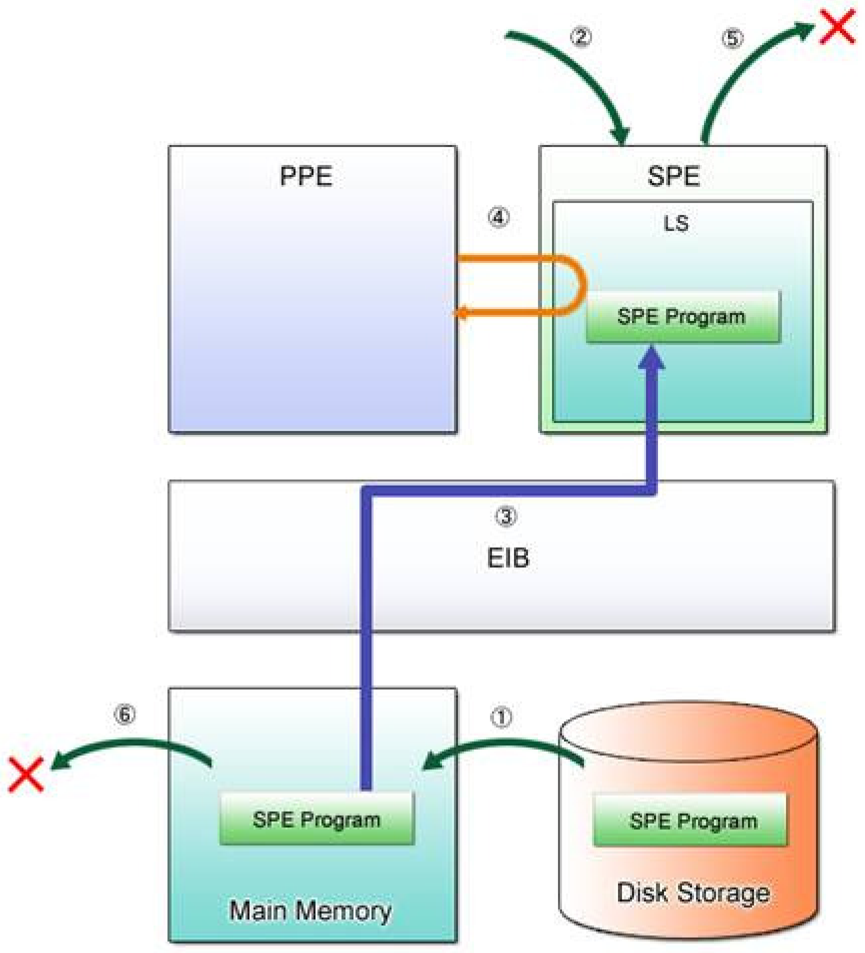

Figure 4.1: SPE Program Execution Sequence. Illustration taken from the Cell Programming Primer [5]

This execution sequence is illustrated in figure 4.1.

One important thing to note about the function *spe_context_run*, is that it is a synchronous function; The call does not return until the program completes execution. This is one of the reasons we have several worker threads managing scheduling, as described in section 3.4.3.

4.2.2 Scheduler initialization

As argued in section 3.4.3, our scheduling implementation requires the SPE contexts to be accessible from all worker threads, allowing them to pick the most suitable context for the workload. The data structure shown in listing 4.6 contains the SPE contexts, which are initialized during the setup routine. The listing is from the scheduler implementation, but is modified for readability.

```
 1 typedef struct _spe_context_handle_t {
 2   spe_context_ptr_t spe;
 3   spe_program_handle_t *prog;
 4
 5   int free;
 6   int spe_num;
 7
 8   char *elf_img;
 9   int img_match;
10 } spe_context_handle_t;
11
12 spe_context_handle_t *spes[NUM_SPE];
```

Listing 4.6: SPE contexts

As we can see in listing 4.6, line 2 contains the pointer to the actual SPE context, as described in section 4.2.1. This pointer is assigned a context with *spe_context_create*, and is used later by the worker threads. On line 3 we have the SPE program handle for the context, which the binary of the SPE program is loaded to. Further, *free* is a flag used by the scheduler to check if the context is available. *spe_num* is an identifier for the context. *elf_img* is a string containing the current image loaded to the context. *img_match* is a flag set by the scheduler, as an indicator of a new program image load is required. The desired number of SPE contexts is defined by *NUM_SPE*, which in our case is six, since our scheduler runs on PS3 machines.

4.2.3 Connection Manager

The scheduler must be able to manage new incoming connections, and to keep reading from the connections we already have. When creating a socket descriptor with *socket*, the kernel by default sets the socket to blocking mode, effectively causing functions like *accept* or *read* to sleep the thread until data arrives. Therefore, using blocking sockets is not an option. Upon creation of a socket, we have the option of making it non-blocking, effectively making it poll the socket for data. However, this type of busy-waiting may potentially hog the PPE, wasting a lot of resources. We have therefore chosen to use the mechanisms of *select*, as it allows us to multiplex incoming connections and data, thus avoiding synchronization primitives between process instances, as forking would lead to.

The scheduler's connection manager sets up a listener in *select*-mode, and waits for data to arrive. Incoming data may either be workloads, or SPE program images, which are handled seperately. SPE program images are identified by the *img_check* flag in the header, and handled by the function *store_image*. This function stores the program image locally, and sends an acknowledgement back to the client, so that it may register the image in its look-up table. If the *img_check* flag is not set, the connection manager proceeds to handle it as a workload.

Before queueing, workloads are assigned a socket number, further used by the worker threads to send results back to the client. If it is required by the workload, memory is also allocated for output data according to the workload's *out_data_size* field. The workload is then queued by calling *queue_workload*, and the connection manager continues to read from the socket.

4.2.4 Workload Queue

The workload queue is implemented as a FIFO buffer, with functions to push and pop workloads from the queue. To avoid race conditions between the connection manager and worker threads, which are both trying to access the queue at the same time, the queue is protected by a mutex.

Workloads are queued with the function *queue_workload*, called from the connection manager upon new requests from the client. To access the queue, this function needs to acquire the associated lock. Once the lock has been acquired, the workload is pushed to the tail of the queue, since we consider all workloads to have equal priorities. The queue also has an associated counter, which is used together with a condition variable by the worker threads. This counter is incremented, before releasing the queue lock. Once the lock is released, the functions broadcasts on a condition variable, waking up the worker threads, and signaling them of a new workload ready for processing.

4.2.5 Worker threads

The worker threads are spawned by the main setup routine, where one worker is spawned for each SPE context created. As mentioned in section 4.2.4, the workload queue has a counter with an associated condition variable, which the worker threads initially sleeps on. When a workload enters the queue, the worker threads are woken up by a signal. Each thread then re-acquires the mutex, checks the queue counter, and pops a workload from the queue if available. Workloads are acquired by calling *queue_get_first*, which returns a pointer to the first element in the FIFO buffer.

The worker thread then needs to acquire an SPE context, which is done by calling *get_free_spe* with the workload as argument. The contexts, as described in section 4.2.2, are protected by a mutex to avoid race conditions. As argued in section 3.4.3, *get_free_spe* tries to find the most suitable context for the workload, thus, checking for a free context with a matching SPE program image first. Should the algorithm succeed at this pass, the *img_match*-flag in the context handle is set, indicating a matching context has been found. If no matching context was found, the algorithm searches for a free context at second pass, where no SPE program image is yet to be loaded. At third pass, the first free context is chosen, regardless of the SPE program image loaded.

The function *get_free_spe* returns a context handle, where we check the *img_match* flag for a matching SPE program image. If the flag is not set, the image must be loaded to the context. This is done by calling the functions *spe_image_close*, *spe_image_open* and *spe_program_load*, respectively, from the libspe2 API. If a new image was loaded, the context handle for the SPE is also updated with the new image.

When the SPE program image is loaded to the context, we can execute the workload on the given SPE. Since we do not load several SPE program images to the LS at once, the *entry*-point is always set to *SPE_DEFAULT_ENTRY*. When the execution has completed, we free the context, wrap the output data with a small header, and send the output back to the client.

4.3 Summary

In this chapter, we presented the implementation details surrounding the runtime library and Cell BE scheduler. In the following chapters, we move on to see how the framework performs in an actual application, using it to encode video with the M-JPEG algorithm in chapter 5, and encrypt data with the AES block cipher in chapter 6.

Chapter 5

Motion JPEG Video Coding

In this chapter, we use our framework to encode video in the Motion JPEG format. We offload the DCT and quantization steps of the algorithm to the Cell BE schedulers, and measure throughput gained for each node added. Video encoding is highly data bound, so network latency may affect the throughput of our application.

This chapter is organized as follows; In section 5.1 we present an overview of the Motion JPEG video codec. Section 5.2 covers our adaption of the codec, and the benchmarks of our implementation are presented in section 5.2.2. In section 5.3, we look at lessons learned.

5.1 Background information

In Motion JPEG, often abbreviated M-JPEG, each video frame is compressed separately using the JPEG still image compression standard. Since each frame is processed separately, the codec does not use any motion estimation or compensation. This leads to very few data dependencies, and it is thus highly parallelizable.

5.1.1 Color space conversion

The first step of M-JPEG coding is usually converting the image from the RGB color model, to a different color space called YCbCr. YCbCr consists of three components;

The luma component Y, represents the brightness of the pixel. The chrominance components Cb and Cr, represents the color of the pixel, in blue and red difference, respectively. The JPEG File Interchange Format (JFIF) usage of JPEG allows YCbCr, where Y, Cb and Cr have the full 8-bit range of 0-255. All 256 levels can be computed directly from 8-bit RGB as follows [60]:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= -0.1687R - 0.3313G + 0.5B + 128 \\ Cr &= 0.5R - 0.4187G - 0.0813B + 128 \end{aligned}$$

5.1.2 Downsampling and block splitting

The conversion to the YCbCr color space enables the first compression step, which is to reduce the spatial resolution of the Cb and Cr components, referred to as "downsampling" or "chroma subsampling". Due to the receptors in the human eye being less sensitive to hue and color saturation than luminance, we can compress images more efficiently by storing more luminance detail (Y component) than color detail (Cb and Cr components).

The downsampling scheme for JPEG is most commonly expressed as a three part ratio, where 4:4:4 has no downsampling, 4:2:2 reduces by a factor of 2 in the horizontal direction, and 4:2:0 reduces by a factor of 2 in both the horizontal and vertical direction. These ratios are illustrated in figure 5.1.

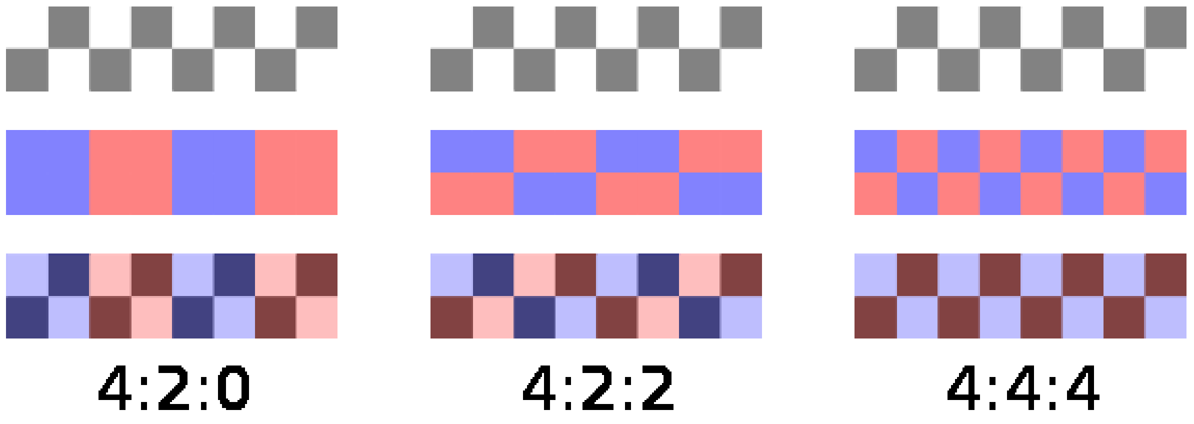

Figure 5.1: Common downsampling ratios for JPEG. Modified illustration from Wikipedia [6].

5.1.3 Discrete Cosine Transform

Next, each component of the image (Y, Cb and Cr) are split into 8x8 pixel macroblocks. Each macroblock is then converted to a frequency-domain representation, using a normalized, two-dimensional Discrete Cosine Transform (DCT).

For an 8-bit image, an 8x8 macroblock might consist of the values

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix}$$

As we can see from matrix, the pixel values from the original block falls in the range $[0, 255]$. Since the DCT is designed to work on values $[-128, 127]$, the mid point of the range is subtracted from each entry, which in this case is 128. This produces a data range that is centered around zero, like the following matrix.

$$\begin{array}{cc} \overset{x}{\longrightarrow} & \\ \begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix} & \Bigg\downarrow y \end{array}$$

The next step is to apply the two-dimensional DCT to the matrix, given by

$$G_{u,v} = \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} g_{x,y} \cos\left[\frac{\pi}{8}\left(x + \frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y + \frac{1}{2}\right)v\right]$$

where $G_{u,v}$ is the DCT at coordinates (u,v), u is the horizontal spatial frequency [0,8>, v is the vertical spatial frequency [0,8>, $g_{x,y}$ is the pixel value at coordinates (x,y), and

α is the normalizing function

$$\alpha_p(n) = \begin{cases} \sqrt{\frac{1}{8}}, & \text{if } n = 0 \\ \sqrt{\frac{2}{8}}, & \text{otherwise} \end{cases}$$

Next, we apply the DCT function above, which yields the following matrix.

$$G = \begin{array}{c c} \overset{u}{\longrightarrow} & \\ \begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -21 & -60 & 10 & 13 & -7 & -8 & 4 \\ -46 & 7 & 77 & -24 & -28 & 9 & 5 & -5 \\ -48 & 12 & 34 & -14 & -10 & 6 & 1 & 1 \\ 12 & -6 & -13 & -3 & -1 & 1 & -2 & 3 \\ -7 & 2 & 2 & -5 & -2 & 0 & 4 & 1 \\ -1 & 0 & 0 & -2 & -0 & -3 & 4 & -0 \\ -0 & 0 & -1 & -4 & -1 & -0 & 0 & 1 \end{bmatrix} & \Big\downarrow v \end{array}$$

The macroblock now consists of 64 DCT coefficients, c_{ij}, where i and j range from 0 to 7. Note the top-left coefficient, c_{00}, which correlates to the low frequencies of the original image block. This is the DC coefficient, while the remaining 63 coefficients are called AC coefficients. The advantage of the DCT is its tendency to aggregate most of the signal in one corner of the results, as may be seen above. The following quantization step accentuates this effect, while simultaneously reducing the overall size of the DCT coefficients, making compression easier in the entropy stage.

5.1.4 Quantization

The human eye is most sensitive to low frequencies, and is good at seeing a change in brightness over a large area. This allows us to reduce the amount of information in the high frequency components.

Subjective experiments involving the human visual system have resulted in the JPEG standard quantization matrix [61], which at quality level 50 is as follows

$$D_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

We then divide each component of the frequency domain by the constants in the quantization matrix, then round to the nearest integer, as given by

$$B_{j,k} = \text{round}\left(\frac{G_{j,k}}{D_{j,k}}\right) \text{ for } j = 0, 1, 2, \cdots, 7; k = 0, 1, 2, \cdots, 7$$

where $G_{j,k}$ is the unquantized DCT coefficients, $D_{j,k}$ is the quantization matrix, and $B_{j,k}$ is the quantized DCT coefficients.

5.1.5 Entropy coding

The macroblock is then further compressed with entropy coding, which involves using a loss-less algorithm to order the components in a zig-zag pattern, also known as run-length encoding (RLE). This groups similar frequencies together, insert length coding zeros, and then uses variable-length Huffman coding on the remains. An illustration of the zig-zag pattern is shown in figure 5.2.

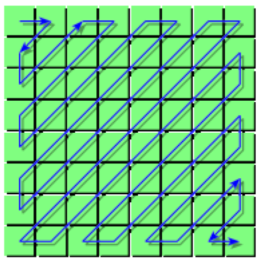

Figure 5.2: Zig-zag pattern. Illustration from Wikipedia [7].

5.2 Adaption for our framework

5.2.1 Implementation

Our implementation focuses on offloading the most compute intensive parts of the M-JPEG algorithm, namely the DCT and quantization process, using a domain decomposition model for work distribution. M-JPEG has few data dependencies, which makes the parallelization strategy simple. Our input consists of a series of images in the YUV format, where each channel is processed separately.

Since the DCT and quantization process can operate on one independent macroblock at a time, we could create workloads containing only a single macroblock. However, since an 8x8 macroblock is only 64 bytes, and DMA supports transfers up to 16kB, this would be very inefficient in terms of both DMA- and network latency. For testing the implementation, we use the 300 frame CIPR foreman sequence [62], which has a resolution of 352x288 pixels. Partitioning the frame to two rows per workload yields $352 * 8 * 2 * sizeof(uint8) = 5632$ bytes input, and $352 * 8 * 2 * sizeof(int16) = 11264$ bytes output, which is inside our DMA range. The work distribution scheme is presented in listing 5.1, and the SPE code is presented in listing 5.2 and 5.3. The listings are from the M-JPEG implementation, but are modified for readability.

In listing 5.1, we call *dct_quantize* for each channel (Y, U, V) in the image. *dct_quantize* creates workloads for the rows in the channel, and dispatches them to the schedulers by calling *cellsched_send*. While waiting for the offloaded workloads, we start writing the output file and define headers, before calling *cellsched_sync* to synchronize the workloads.

```
...
cellsched_setup_hosts(8);
cellsched_add_host(''cell-1.host'');
cellsched_add_host(''cell-2.host'');
...
cellsched_add_host(''cell-8.host'');

static void dct_quantize(uint8_t *in_data, uint32_t width, uint32_t
    height,
  int16_t *out_data, uint32_t padwidth,
  uint32_t padheight, int channel)
```

```
{
  nrows = height/8;
  ...

  while(i = 0; i < nrows; i++) {

    workload_t load;
    load.elf_image = ''spe_dct.elf'';
    load.in_data = in_data+bufpos;
    load->out_data = out_data+bufpos;
    load->in_data_size = (width*8*2)*sizeof(uint8_t);
    load->out_data_size = (width*8*2)*sizeof(int16_t);
    load->dtype = DSHORT;
    load->uint32_arg[0] = width;
    load->uint32_arg[1] = channel;
    bufpos += width*8*2;

    cellsched_send(load);
  }
}

dct_quantize(image->Y, width, height, out->Ydct, ypw, yph, YCHAN);
dct_quantize(image->U, (width*UX/YX), (height*UY/YY), out->Udct, upw, uph, UCHAN);
dct_quantize(image->V, (width*VX/YX), (height*VY/YY), out->Vdct, vpw, vph, VCHAN);

write_SOI();  // Write start of frame
write_DQT();  // Define Quantization Table(s)
write_SOF0(); // Start Of Frame 0(Baseline DCT)
write_DHT();  // Define Huffman Tables(s)
write_SOS();  // Start of Scan */

cellsched_sync();
...
```

Listing 5.1: M-JPEG work distribution

In listing 5.2, we calculate the DCT coefficients. Both the DCT and quantization steps are implemented using SIMD-instructions, to speed up the processing. On line 4, we subtract 128 from the input to center the values around zero, as explained in section 5.1.3. We then use a pre-calculated look-up table when applying the two-dimensional DCT to the matrix, as done on line 5 to 7. Last, we add the values to get the DCT coefficients on line 8.

```
 1  ...
 2  for(u = 0; u < 64; ++u) {
 3    for(j = 0; j < 8; ++j) {
 4      vcoeff = spu_sub(vin_float[(j*width+x)>>2], spu_splats((float)
           128.0f));
 5      vfdct[0] = spu_madd(vcoeff, vcos_tab[t++], spu_splats((float)
           0.0f));
 6      vcoeff = spu_sub(vin_float[((j*width+x+1)>>2)+1], spu_splats((
           float)128.0f));
 7      vfdct[1] = spu_madd(vcoeff, vcos_tab[t++], spu_splats((float)
           0.0f));
 8      dct[u] += (fdct[0] + fdct[1] + fdct[2] + fdct[3] + fdct[4] +
           fdct[5] + fdct[6] + fdct[7]);
 9    }
10  }
```

Listing 5.2: DCT computation

Listing 5.3 shows the quantization of the matrix. On line 3, we use a pre-calculated look-up table to normalize the data. We then divide the DCT coefficients by the quantization values, add 1/2, and floor the sum, on lines 4 to 6.

```
1  for(v = 0; v < 16; ++v) {
2    vdct[v] = spu_madd(vdct[v], vnorm_tab[v], spu_splats((float)0.0f)
        );
3    vsumf = _divf4(vdct[v], vquantf[v]);
4    vsumf = spu_add(spu_splats((float)0.5f), vsumf);
5    vsumf = _floorf4(vsumf);
6  }
```

Listing 5.3: Quantization

5.2.2 Evaluation

The M-JPEG implementation was benchmarked using a setup consisting of an Intel Core i5 2.67 GHz with 4 cores as the client, and 8 Sony PS3s running the scheduler on Linux, connected via a gigabit local area network. The test consisted of encoding the 300 frame CIPR foreman sequence, and measuring the elapsed time. The tests were performed 5 times for each setup, and the average values are presented in this section.

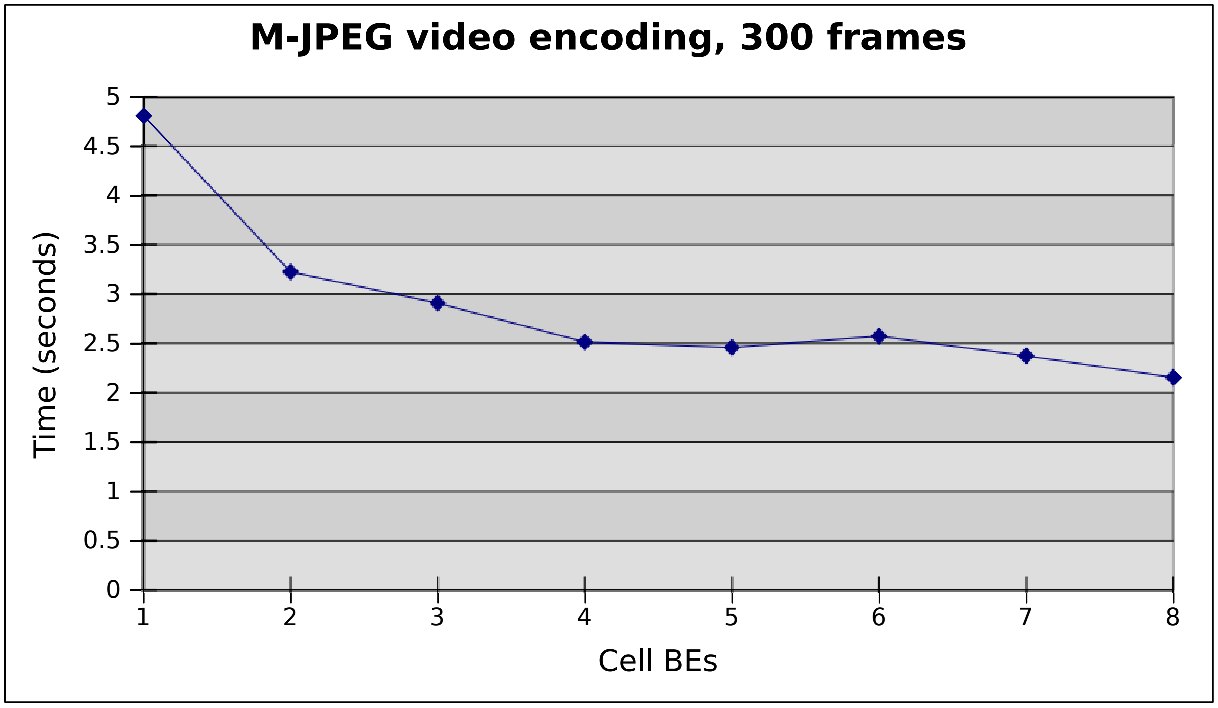

Figure 5.3: A benchmark of M-JPEG video encoding using one to eight Cell BEs.

Figure 5.3 graphs the performance of the M-JPEG implementation with regard to the number of nodes used by the client. The most important observation from the graph, is the limited scalability of added nodes. We compared our results to an M-JPEG implementation optimized to run directly on a single Cell BE, using mailboxes and double buffering for workloads. The optimized implementation had an average runtime of 4.19 seconds, while our implementation had an average runtime of 4.81 seconds with a single Cell BE. However, with an addition of 7 Cell BEs, we only gained a performance increase of 2.86 seconds.

Since the implementation offloads the DCT and quantization for a frame at a time, these results are most likely caused by the rate of which we dispatch workloads. As mentioned in section 5.2.1, each workload is limited to two rows of data, yielding 36 workloads per frame. We can thus only dispatch bursts of 36 workloads before needing to synchronize, and do post-processing for the frame. The 300 frame CIPR foreman sequence has a total size of 46 MB. With our best average run of 2.16 seconds using 8 Cell BEs, we get a throughput of approximately 170 Mbps, which is far from our bandwidth limit of 1 Gbps. While overhead from the runtime library is likely to affect the dispatch rate, due to the pattern of workload bursts, overhead from network latency might additionally influence the throughput.

Figures 5.4 to 5.6 graphs each node's workload queue, in three different setups, during the encoding process. When using a single Cell BE, we are able to saturate the SPEs, reducing idle time to a minimum. However, when adding more nodes, the graphs clearly show SPE starvation. Since we actually have more available SPEs than concurrent workloads (48 SPEs for 8 Cell BEs), this indicates that workloads are being processed faster than the client is able to dispatch. However, we also observe that the graphs tends to spike at certain points during execution. We believe this is due to the tests being performed in a non-sterile, trafficated network, and the time consumed by the workloads on the SPEs is so low, that the performance is dictated by workload management and data transfers.

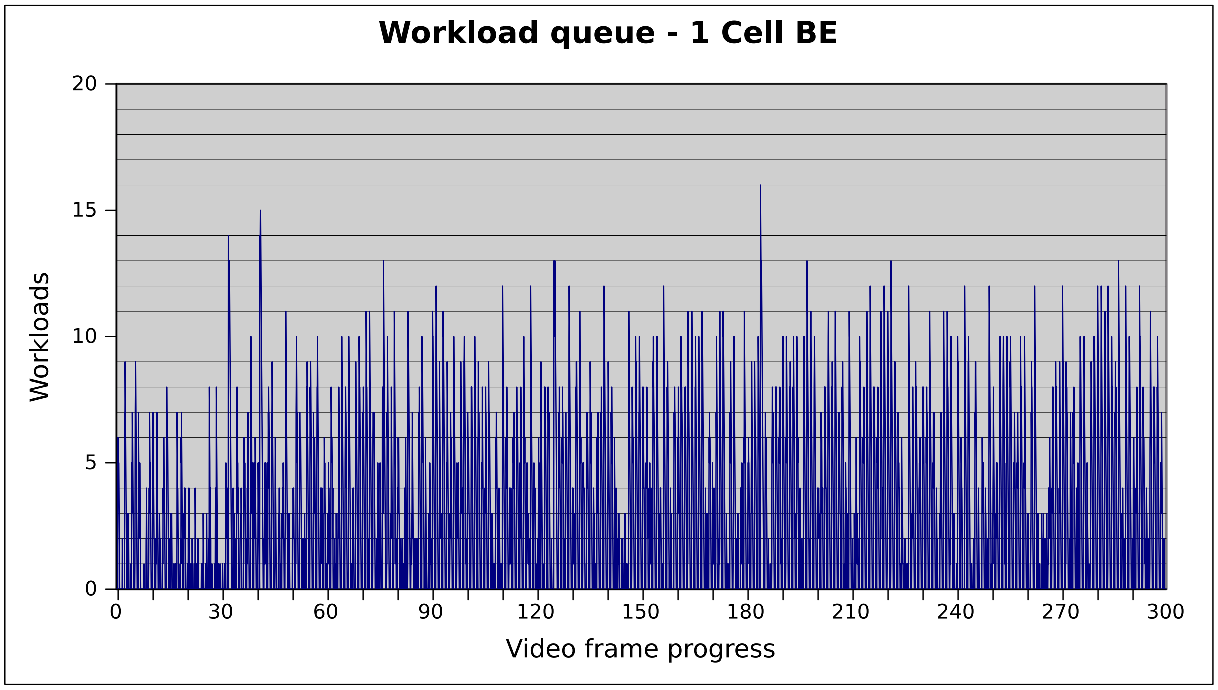

Figure 5.4: Scheduler queue during encoding, client offloading to 1 node.

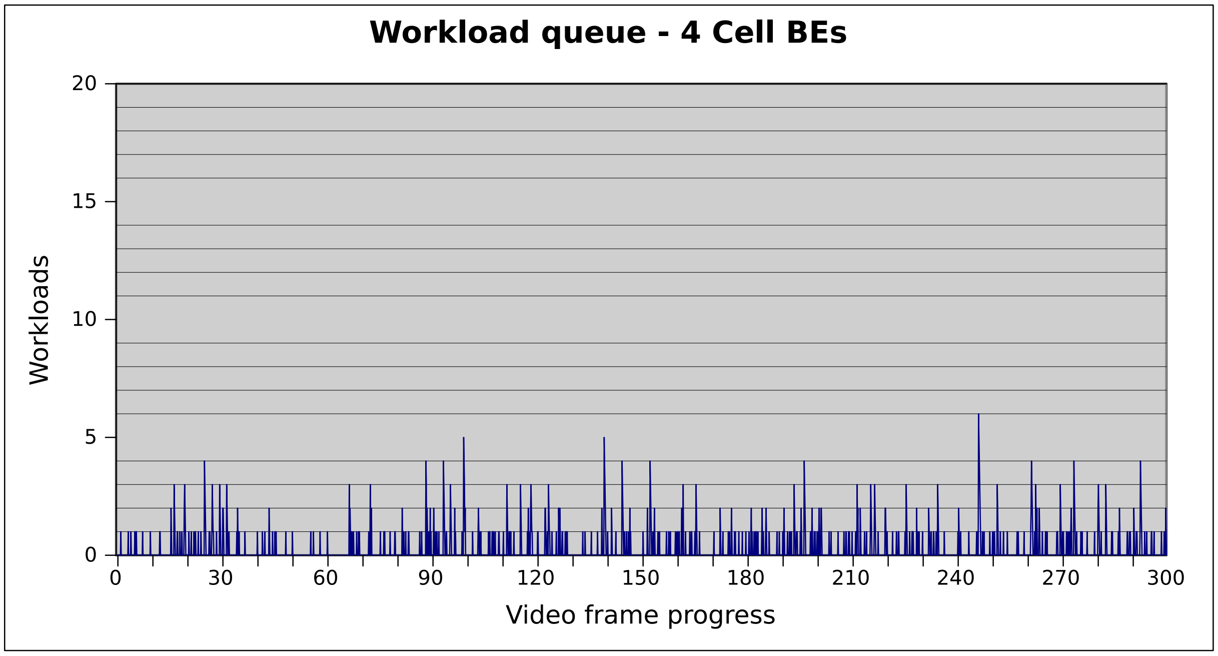

Figure 5.5: Scheduler queue during encoding, client offloading to 4 Cell BEs.

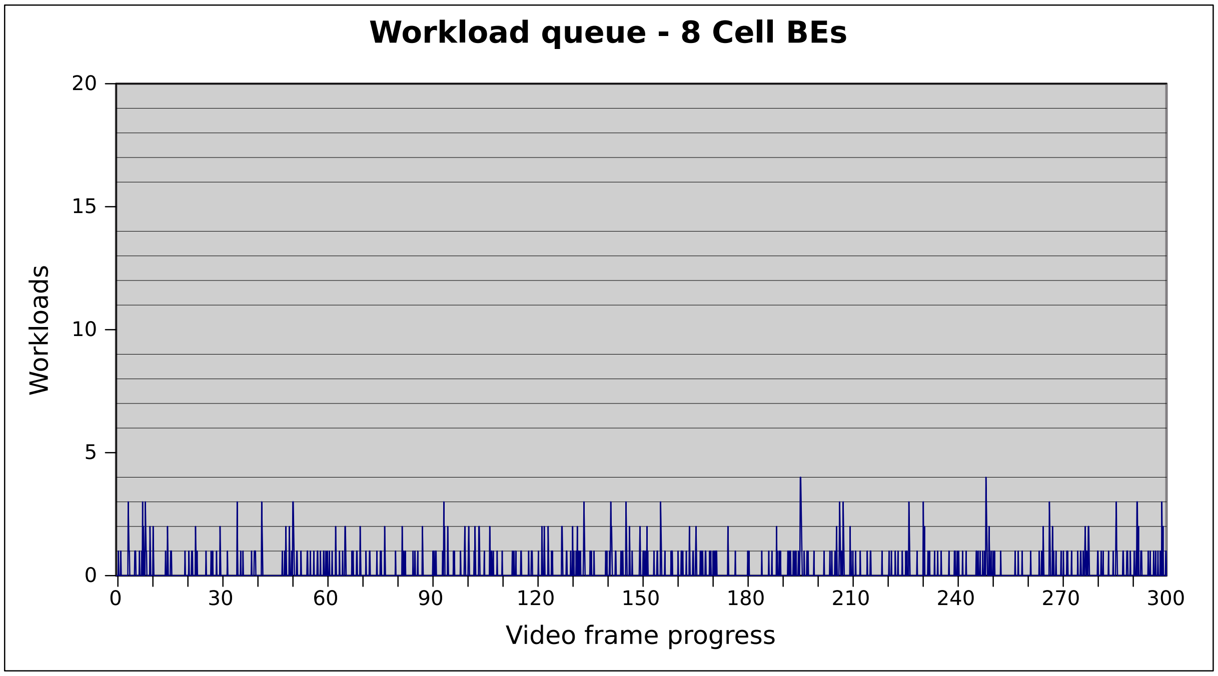

Figure 5.6: Scheduler queue during encoding, client offloading to 8 Cell BEs.

5.3 Lessons learned

The results indicate that scheduler performs well, and the throughput is restricted by the rate, or the pattern, of which we dispatch workloads, exposing overhead in the runtime library.

As workloads are being processed faster than the client is able to dispatch, this puts further pressure on the runtime library and data transfers. The limitations in the runtime is likely to be caused by dispatch overhead from workload creation, and synchronization mechanisms in the receiver threads. To be able to efficiently utilize all 8 Cell BEs, and to be able to hide potential network latency, we must provide a steadier stream of workloads, or dispatch in greater quantities to saturate the SPEs. This is hard to do with the M-JPEG implementation, due to the post-processing required after each frame.

The implementation might be improved further by handling each frame in a separate thread, and by offloading additional parts of the algorithm to the nodes, but due to time constraints, this is left as further work.

By evaluating a data-bound algorithm, where time consumed on the SPEs is low, we get a good indication of how workloads need to justify overhead from the runtime library. Although the scalability for this implementation is limited, the performance compared to the Cell BE optimized version is good. Provided a work distribution scheme, the runtime library API is also easy to use, requiring only a minimum of configuration.

5.4 Summary

In this chapter, we have given a brief overview of Motion JPEG video encoding, and provided a parallel implementation using our framework. The DCT and quantization steps of the algorithm was offloaded to Cell BEs, and we benchmarked the throughput for each added node. While the throughput gained by a single node is similar to an optimized implementation using mailboxes and double buffering, our implementation suffers from limited scalability. The scheduler performed well, indicating that further optimizations should be done in the runtime library.

Chapter 6

Advanced Encryption Standard

In this chapter, we use our framework to parallelize the Advanced Encryption Standard algorithm. Our main focus is on the dispatching of workloads, and to see how well our implementation scales when imposing a heavier load on the SPEs. As the AES algorithm itself is not our main focus, we base our work on an existing single threaded implementation for the x86 architecture, and adapt the code to our framework.

This chapter is organized as follows; In section 6.1, we present an overview of the AES algorithm and modes of operation. In section 6.2 we introduce the implementation we base our work on, and the adaption for this implementation is shown in section 6.3. The benchmarks can be found in section 6.3.2, and the lessons learned are presented in section 6.4.

6.1 Background information

The Advanced Encryption Standard (AES) [63] is a symmetric-key block cipher developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen. The standard is also known as Rijndael, which was its title when submitted during the AES selection process.

On January 2, 1997, The National Institute of Standards and Technology (NIST) announced that they needed a successor to the Data Encryption Standard (DES) [64]. This was because DES has a relatively small 56-bit key, which was becoming vulnerable to brute force attacks. The small key size problem was solved by Triple-DES (3DES), which applies the DES cipher algorithm three times to each data block, effectively protecting it against such attacks. However, DES was primarily designed for hardware, and is relatively slow when implemented in software, making it unsuitable for limited-resource platforms.

In the nine months to come, fifteen different designs were submitted by interested parties from several countries. The candidates were investigated by cryptographers, and were assessed on security, performance, and on their feasibility in limited environments. Two conferences was held by NIST to discuss the submissions, and in August 1999, they announced that the fifteen submissions had been narrowed down to five. A third conference was held in April 2000, where the finalists were further analyzed, and a presentation was held from representatives from each of the teams. The final announcement was made on October 2, 2000, where Rijndael had been selected as the winner. AES was approved as FIPS PUB 197 [63] by November 2001.

6.1.1 AES Cipher overview

The AES standard specifies the Rijndael algorithm, a symmetric-key block cipher that uses a fixed block size of 128 bits, and key sizes of 128, 192 of 256 bits. Because of the fixed block size, AES operates on 4x4 array of bytes, referred to as the *state*.

The AES cipher is specified as a number of repeating transformations, called rounds. Determined by the size of the cipher key, 10, 12 or 14 rounds with the following steps are performed:

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

Before these steps can be applied, there is an initial step called *Key Expansion*, also referred to as Rijndael key schedule. In this step, the cipher key is expanded into unique keys for every round, i.e., the round key. Excluding the first and last round, all steps are performed each round. In the initial round, only the *AddRoundKey* step is performed. In the last round, all steps but *MixColumns* are performed. Following, the steps are explained in detail.

Substitute Bytes

In the *Substitute Bytes* transformation step (figure 6.1), each byte in the array is updated using an 8-bit substitution box, also referred to as the Rijndael S-box. This transformation provides non-linearity in the cipher, and is important to prevent cryptographic attacks based on algebraic properties.

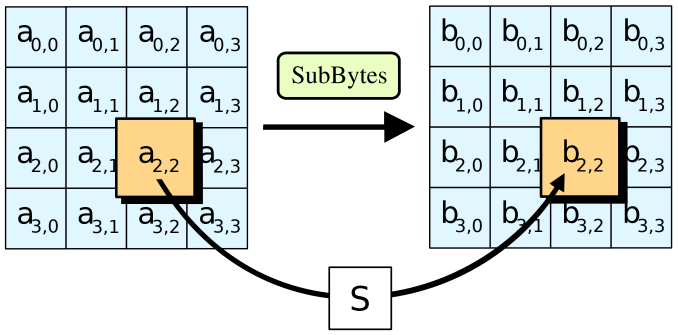

Figure 6.1: Substitute Bytes. Illustration from Wikipedia [8].

Shift Rows

The *Shift Row* transformation (figure 6.2) operates on the rows of the state, shifting the bytes in each row by a certain offset. The first row is left unchanged. Each byte in the second row is shifted one to the left, and the third and forth row are shifted by offsets of two and three, respectively. This transformation makes each column of the output state contain bytes from all columns of the input state.

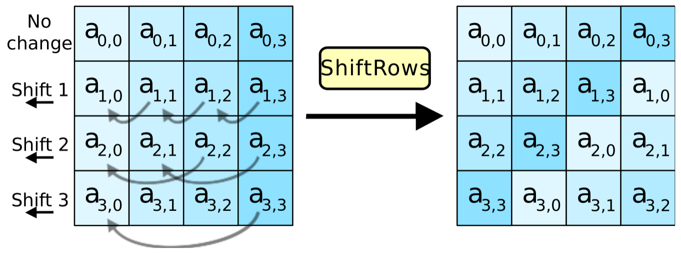

Figure 6.2: Shift Rows. Illustration from Wikipedia [8].

Mix Columns

The third transformation, *Mix Columns* (figure 6.3), combines four bytes of each column of the state. The transformation takes four bytes as input, and outputs four bytes, where each input byte affect all four output bytes. Together with shift rows, this transformation provides diffusion in the cipher.

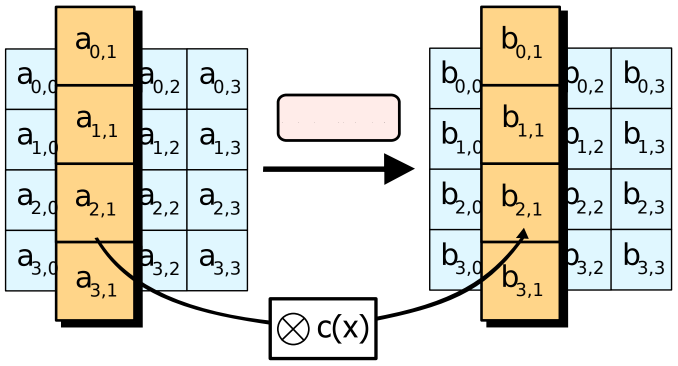
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\otimes c(x)$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 6.3: Mix Columns. Illustration from Wikipedia [8].

Add Round Key

In the *Add Round Key* step (figure 6.4), each byte of the round key, derived from the cipher key in the *Key Expansion* step, is combined with each byte of the state using bitwise XOR.

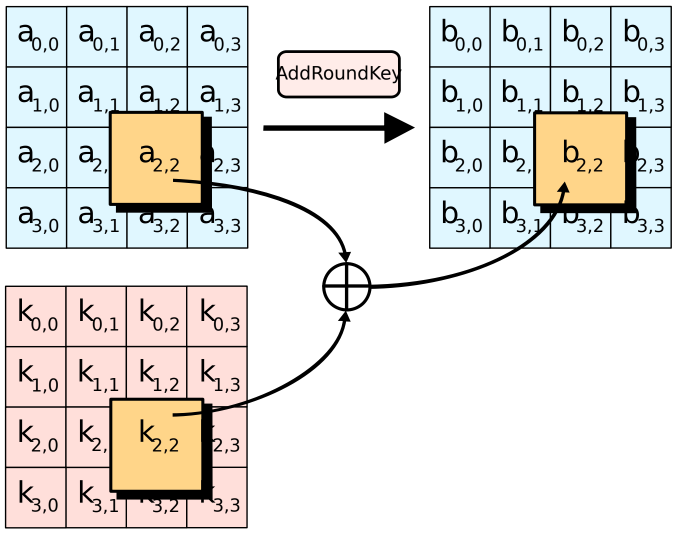
AddRoundKey
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,2}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,2}$
$k_{3,0}$ $k_{3,1}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,3}$
$b_{2,0}$ $b_{2,2}$
$b_{3,0}$

Figure 6.4: Add Round Key. Illustration from Wikipedia [8].

6.1.2 Modes of operation

A block cipher by itself only supports encryption of a single data block, equal to the cipher's block size. When data is of arbitrary length, it must first be partitioned into separate cipher blocks, and since block ciphers work on fixed size units, the last block is usually extended using a suitable padding scheme. A mode of operation describes the process of encrypting each of these data blocks.

The most common modes of operation include *electronic codebook* (ECB), *cipher-block chaining* (CBC), and *counter* (CTR). The simplest of the modes is ECB, where the data is divided into blocks and processed separately, making it easy to parallelize. However, the drawback with this mode is that identical plaintext blocks are encrypted into identical ciphertext blocks, which creates a pattern in the data, as illustrated in figure 6.5. In CBC mode, the plaintext of each block is XORed with the previous ciphertext block, making each ciphertext block dependent of all plaintext blocks up to that point. To make each pass unique, an initialization vector (IV) is used in the first block. CBC's main drawback, is that it cannot be parallelized due to the dependencies of previous blocks. Also, a single bit error in a block affects all following blocks, making them undecryptable.

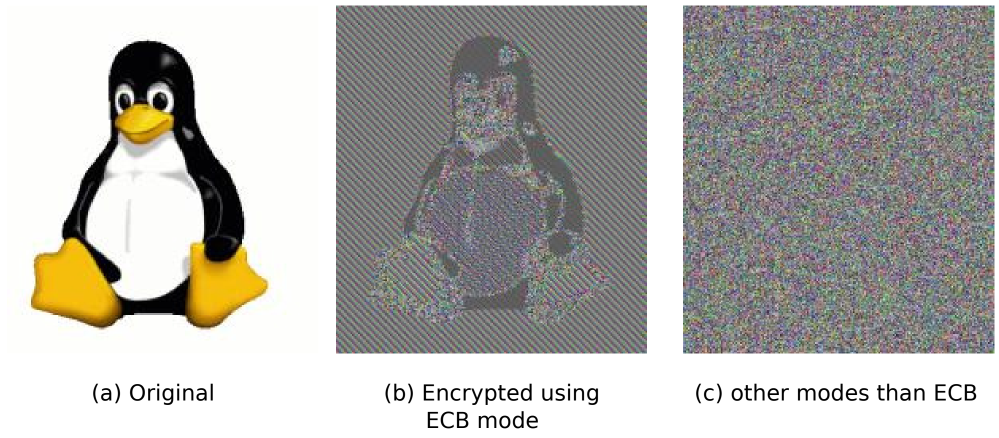

Figure 6.5: Comparison of encryption using different modes. Illustration from Wikipedia [9].

In CTR mode, instead of encrypting the plaintext directly, a counter is encrypted to generate a series of pseudo-random output blocks. The counter is initialized with a nounce, much like an IV, and is incremented for each block using a shared key. The output block is then XORed with the plaintext, resulting in the ciphertext. This allows for blocks to be processed independently, and bit errors does not affect following blocks. Also, because of the properties of the XOR operation, the same algorithm can be used for both encryption and decryption. Based on these advantages, we have chosen CTR as the mode of choice for our implementation. CTR mode is illustrated in figure 6.6.

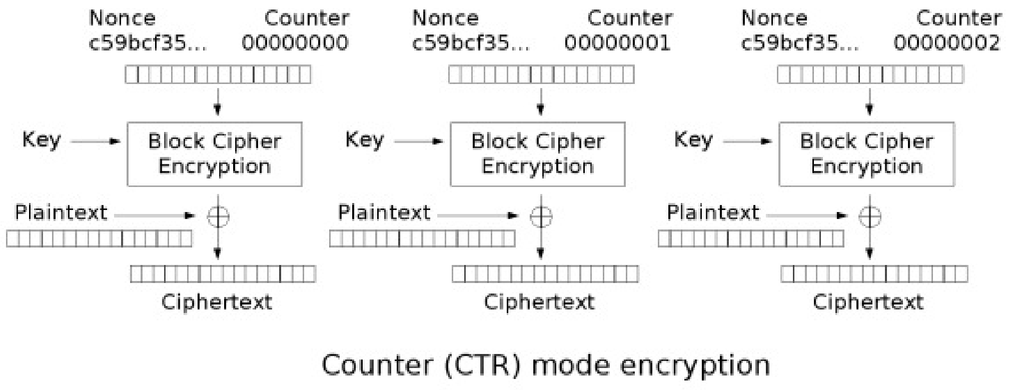

Figure 6.6: Counter mode (CTR) block encryption. Illustration from Wikipedia [9].

6.2 Software basis

Our AES implementation is based on a public domain implementation written by Philip J. Erdelsky [65], optimized for single core x86 usage. We used a modified version of this code ([66]), written by Håvard Espeland at Simula Research Laboratory, optimized for the Cell BE. The implementation uses pre-defined look-up tables, 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. Due to large look-up tables, and many load/store operations, the implementation does not exploit SIMD capabilities.

6.3 Adaption for our framework

6.3.1 Implementation

The implementation uses pre-defined look-up tables to increase the efficiency of the cipher. With this implementation, we want to see how well our framework scales when imposing a heavier load on the SPEs, compared to the M-JPEG implementation, which is heavily data-bound. To make the AES algorithm even more compute-bound, we decided to encrypt each block a hundred times, effectively making the workload consume more time. Our implementation scales to any number of concurrent workloads, where each workload contains 16 kB of plaintext.

As workloads are processed separately by the nodes, round keys must be calculated for each set of blocks. Since DMA transfers are asynchronous, we can explicitly parallelize computation and data transfers. Calculating round keys and encrypting the counter values are more time consuming than transferring the plaintext, so we put the DMA transfer in the background, and make sure the plaintext transfer is finished before XORing it with the encrypted counter values.

6.3.2 Evaluation

We benchmarked the AES implementation using the same setup as with M-JPEG, having the client running on a Intel Core i5 2.67 GHz with 4 cores, and 8 Sony PS3s running the scheduler on Linux, connected via a gigabit local area network. We tested the implementation by encrypting 128 MB of data, and measuring the elapsed time. The tests were performed 5 times for each setup, and the average values are presented in this section.

Figure 6.7 shows the performance of the AES implementation with regard to the number of Cell BEs used by the client, and the number of workloads dispatched between each synchronization. When dispatching bursts of 256 or more workloads, the setup scales well, showing a near linear increase in throughput. As discussed in chapter 5, the framework seems to have limited scalability when the number of workloads is low, which also reflects in this implementation. Since no post-processing is done after each set of workloads, overhead is exposed more clearly, indicating that the bottleneck most likely lies at the runtime library. However, due to the parallelization strategy of the AES algorithm, the implementation scales to any number of workloads, allowing us to fully saturate the nodes, which in turn reduces SPE starvation.

The large difference in scheduler load with regard to workload queue, and number of concurrent workloads, is graphed in figures 6.8 and 6.9. With small bursts of workloads, processing is faster than the client can dispatch, leaving the SPEs idle between client synchronizations. However, by increasing the number of concurrent workloads,

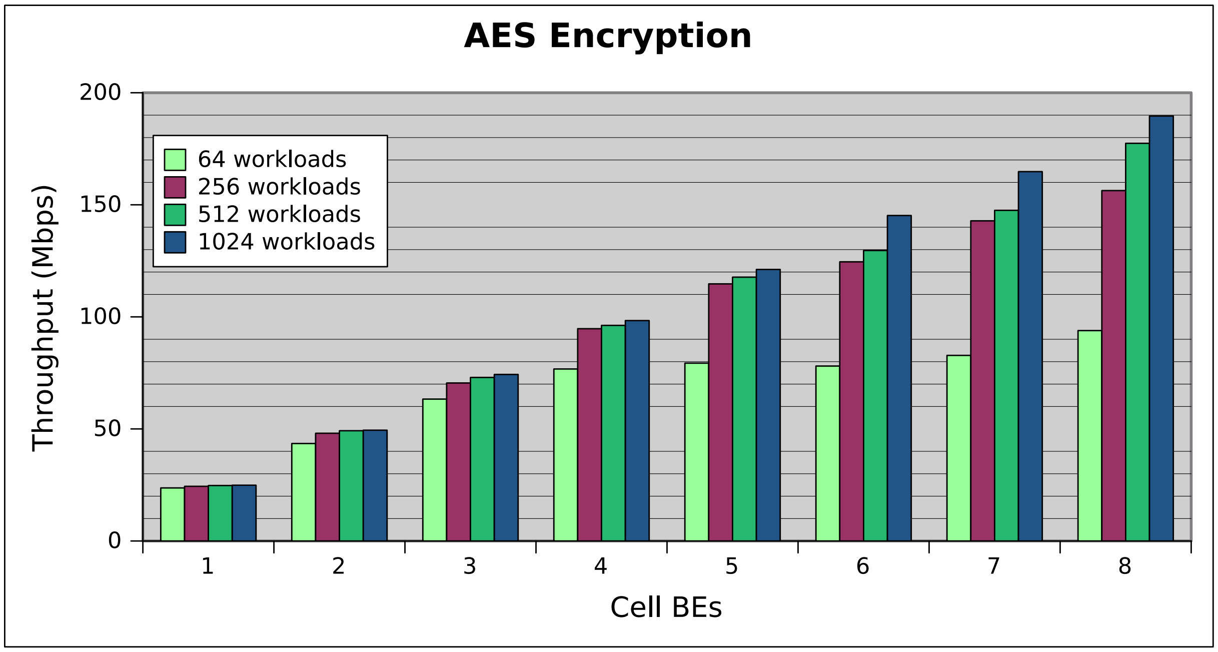

Figure 6.7: A benchmark of AES with increasingly added nodes.

we are able to saturate the nodes, reducing overhead from the runtime library to a minimum. As observed from figure 6.9, the queue drops to zero after each burst of workloads, which is due to workloads being dispatched and synchronized by a single thread.

6.4 Lessons learned

The AES implementation scales well when increasing the number of workloads. Due to the blocks being completely processed on the SPEs, the work distribution strategy allows us to create any number of jobs, reducing the number of synchronizations needed by the client, which in turn cuts down on framework overhead. The workloads also consume more time on the SPEs, which seems to reduce dispatch overhead. For 256 workloads and above, the implementation provides a near linear increase in throughput.

Each workload only occupies a small amount of the LS on each SPE; Thus, further improvements could be done to the scheduling algorithm, as discussed in section 3.4.3. By either filling the LS with workloads prior to execution, or by implementing double buffering mechanisms available to the SPE program, workload scheduling can be

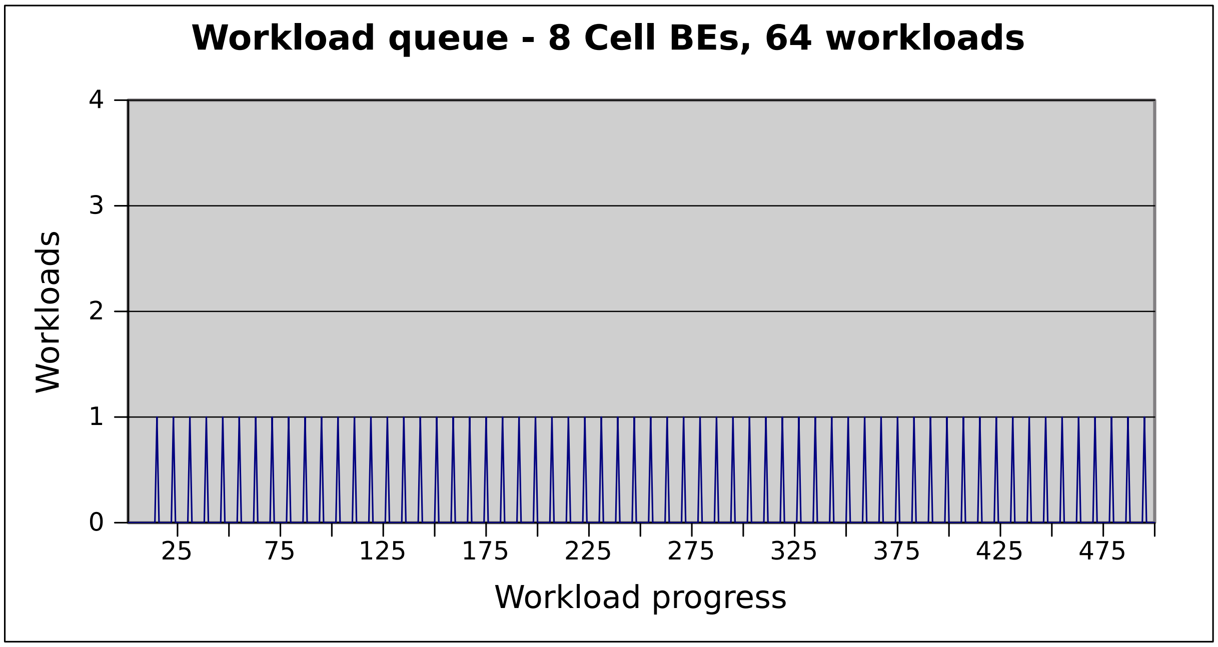

Figure 6.8: Scheduler queue during encryption, client offloading to 8 Cell BEs, with 64 workloads each pass.

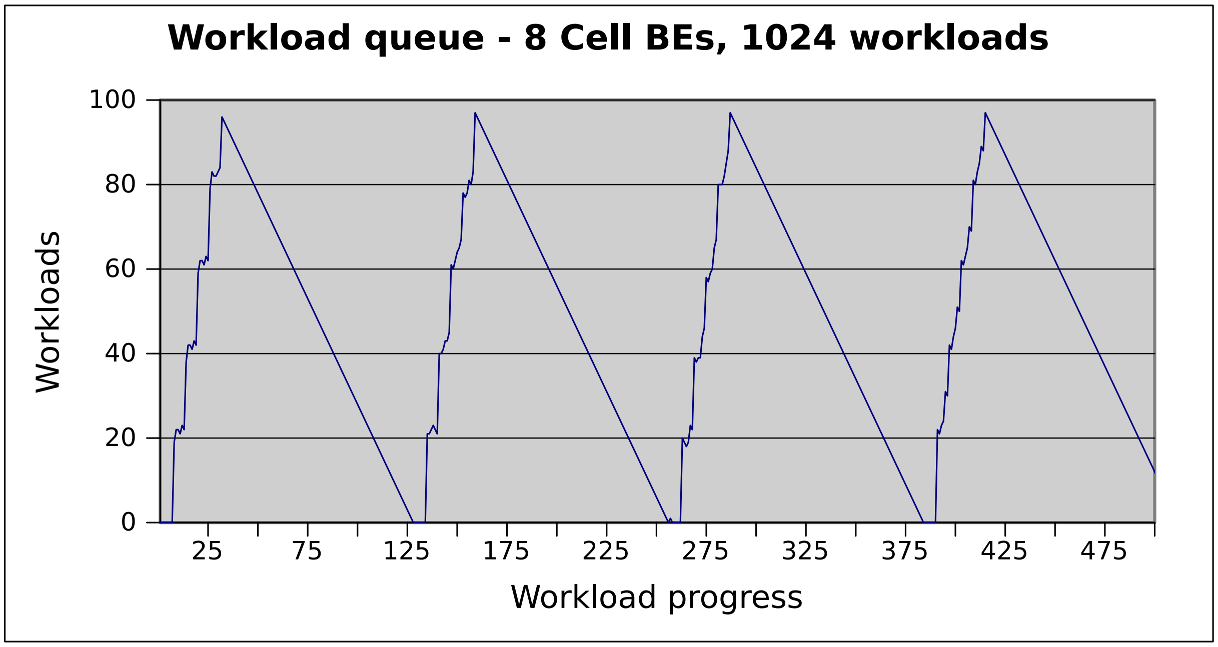

Figure 6.9: Scheduler queue during encryption, client offloading to 8 Cell BEs, with 1024 workloads each pass.

further improved. But due to time constraints, this is left as further work.

6.5 Summary

In this chapter, vi have given a brief overview of the Advanced Encryption Standard, and provided a parallel implementation using our framework. The implementation was evaluated by measuring the encryption of 128 MB data, and testing the throughput gained for each added node. We split the data into workloads of 16 Kb plaintext, and performed the encryption on the SPEs. The system showed good scalability for 256 workloads and above.

Chapter 7

Discussion

In this chapter, we discuss the results and lessons learned from the M-JPEG and AES implementations, and propose possible solutions where the framework did not perform optimally. We also discuss the future of the framework, and the potential possibilities it may provide.

7.1 Results and performance

While the M-JPEG implementation showed good performance offloading to a single node, compared to a more optimized version running directly on the Cell BE, it suffered from limited scalability when offloading to multiple nodes. When inspecting the workload queue in the scheduler, it is clear that the workloads, in terms of time consumed on the SPEs, do not justify the dispatch and synchronization overhead in the runtime. As a result, network latency is likely to have an additional impact on the throughput, since idle time occurs on the connection each time the client is performing synchronization and post-processing of a frame.

The overhead in the runtime might be caused by several reasons: Since the endianess may differ for the client and target machines, both the header and data needs to be converted from host byte order to network byte order. Although these are only bit shift operations, the runtime could benefit from a more serialized way of packing and unpacking workloads. A second possibility is the synchronization mechanism. Like described in section 4.1.4, the function *cellsched_sync* yields the thread if output data from all workloads in the window has not arrived, which in turn allows for one of the receiver threads to continue execution. Although this did provide better performance than busy waiting, it puts the calling thread in the end of the queue for its static priority, which might not be optimal if the runtime is only waiting for a single node. Moreover, if the application relies on frequent synchronizations, as with the M-JPEG implementation, we are unable to provide a steady stream of workloads, which in turn might lead to overhead from network latency. This scenario was confirmed with the AES implementation, by adjusting to a higher rate of synchronizations, which in effect reflected the same results. However, by adjusting to a lower rate of synchronizations, it seems like we are able to hide potential network latency and runtime overhead, thus increasing application throughput. While lowering the number of synchronizations is a way of mitigating overhead, it may not be an option for all applications, where the data distribution strategy restricts the number of concurrent workloads. This in turn leads to low utilization of the Cell BE nodes, and causes starvation on the SPEs. As described in chapter 3, the nodes are capable of handling multiple clients. Although connecting multiple clients will most likely not improve runtime overhead, it may be a way of utilizing the nodes more efficiently. Investigating this is left as further work.

Of the two implementations, the AES block cipher clearly showed the best scalability, having a near linear increase in throughput for each node added. As each independent block is encrypted entirely on the SPEs, the implementation allowed us to provide the nodes with a higher rate of workloads, thus hiding potential network latency. The workloads also consumed more time on the SPEs, compared to the M-JPEG implementation, resulting in the framework being less dictated by data transfers and data management. This was confirmed by inspecting the scheduler's workload queue, which showed much smoother graphs and transitions. However, although we are able to provide more work, resulting in the queue growing much faster than the SPEs can process, this also indicates that improvements need to be done in the scheduler. In the setup using 8 Cell BEs and 1024 workloads, each node is provided $1024/8 = 128$ workloads. In the graph for this setup, we can see the queue peaking at roughly 95 workloads, which tells us by the time the client has dispatched all 1024 workloads, each node has only processed roughly 33 workloads. This is most likely due to the lack of double buffering for the SPEs, which is hard to implement with our current PPE-centric scheduler. However, this would be achievable with an SPE-centric scheduler, like the MARS framework, where workloads are fetched directly by the SPEs. Due to time constraints, this is left as further work.

All the tests were benchmarked on up to 8 Cell BEs. While we would have liked to perform some of the tests on additional machines to investigate further scalability, we did not have more than 8 machines in our lab. Overall, the framework seems to work well for workloads that justify the dispatch and transfer overhead, as with the AES implementation. We have also identified certain limitations in our framework, which addressing is left for further work.

7.2 Applicability

With regard to the usability of the framework, it provides a very simple interface for offloading and does not require a paradigm shift as many other distributed frameworks [13, 16, 54]. The user merely has to define a set of machines running the scheduler, create the workloads to be offloaded, and call a single function, while the runtime handles the rest. This allows for the programmer to focus on the work distribution strategy, and to provide work for the client while the offloaded workloads are processed. Furthermore, contrary to many of the frameworks mentioned in section 2.5.3, our framework has the ability to model iterative algorithms, and supports task-parallelism in workloads.

For the M-JPEG implementation, we offloaded the DCT and quantization part of the algorithm, which both work on a single macroblock at a time. For the AES implementation, we offloaded all steps of the cipher, which work on a 4x4 arrays of bytes. Both of these implementations are embarrassingly parallel problems, considered the lack of data dependencies and communication between workloads. Embarrassingly parallel workloads are usually easy to extract from a given problem, and has the ability to be processed independently. Typically, such problems include brute-force searches, where a workload contains a set of candidates to be tested against certain criteria, and data analysis, where a data set is processed independently with a given algorithm. Specific examples include the Folding@home project [67], the k-means classification algorithm, and the Mandelbrot set. In addition to being embarrassingly parallel, many problems also include iterative algorithms, where the number of iterations cannot be determined prior to execution. The characteristics of these problems fits our framework well, since each workload may run to completion on a predisposed processing element.

However, for applications of other granularities, having a higher degree of data dependencies or requiring communication of intermediate results, the framework may have limited applicability, as it in its current state does not support communication between workloads. Examples of such applications may be more advanced video encoders, like the H.264 video codec, which relies heavily on motion estimation between frames to do compression. We did not consider workload communication for fine and coarse-grained applications as part of our initial design, since the main idea was to provide a simple interface for offloading. Although communication mechanisms would bring the framework to a whole new level of complexity, it is plausible to add as a future extension to the framework. However, this would require re-thinking many aspects of the framework. In the following section, we will briefly look at some of these requirements, and propose possible solutions.

7.3 Future possibilities

During the design and implementation of the framework, there were a lot of ideas and features we would have liked to test out, but were either outside our scope, or restricted by the current availability of other frameworks. In this section, we will discuss some of these ideas, and their possibility and potential in a future version of the framework.

7.3.1 Workload communication

Workload communication introduces a whole new level of complexity to the framework, but also opens up many new possibilities for applications of different granularities. While there are many requirements to fulfill the support of such functionality, we will briefly discuss some of the main challenges.

Communication may be provided at two different levels in the framework: SPE-to-SPE communication, and node-to-node communication. While SPE-to-SPE communication would introduce less overhead, it requires management of workload dependencies across nodes, which will affect the distribution strategy in the runtime. Dependencies will most likely need to be addressed and set up by the user. A possible solution to this, would be introducing workload groups, much like the block/thread relation in CUDA, or the work-group/work-item relation in OpenCL. By defining workloads that communicate inside a group, the runtime may dispatch groups as an entity to nodes, instead of distributing the workloads as individual packages.

To set up synchronization events, the framework will need to provide a message passing interface for SPE programs. As SPEs have support for mailbox and signal-notification messaging, this could be implemented as an abstraction layer on top of these mechanisms. Moreover, the scheduler will need mechanisms to perform context switches, and the ability to store intermediate results. Node-to-node communication adds further complexity to the framework, as workloads will either require a mapping of the current cluster, or the ability to multicast/broadcast synchronization events to other nodes. A possible solution to this scenario would be introducing communication channels between nodes, available as part of the message passing interface for SPE programs.

It should be noted that MARS currently supports task-to-task communication in form of barriers, event flags, semaphores, and signals. If a future version of MARS will support the Cell BE in distributed environments, it might be a viable option to our current scheduler.

7.3.2 Data distribution and compression

As discussed earlier, our framework seems in certain scenarios to be limited by dispatch overhead in the runtime and network latency. We have seen the popularity of distributed file systems in other frameworks [13, 14, 53], providing a fast and efficient way of distributing data. Adapting our framework to be used in conjunction with a distributed file system might be a viable option worth further investigation, as this will relieve the framework of explicit data transfers across nodes. Additionally, it might help mitigate network latency by transferring data in larger quantities, rather than in smaller workload-sized units. However, distributed file systems like Hadoop and GFS are not intended for general purpose applications running on general purpose file systems. They are designed for batch processing of files in the GB to TB range, rather than interactive use, and puts emphasis on high throughput rather than low latency of data access. This limits their potential as an extension to our framework, as we have based our workload size on the restrictions of the Cell BE's DMA engine. Furthermore, data throughput of small file transfers are usually orders of magnitude worse than bulk transfers of larger files. While aggregating and compressing small files may be a solution to the latter problem, packing and unpacking can be slow, which may introduce additional overhead. Moreover, applications may take an arbitrary number of input files of different sizes, making predictions of package sizes difficult.

As a viable distributed filesystem for our framework, we propose investigating options as the Network File System (NFS) [68], or the Chirp [69] filesystem for grid computing. NFS is an RPC based protocol, widely used in clusters of different sizes, designed for block-based access to files over a relatively low latency network. A benefit of NFS is its simplicity, and efficiency of small random accesses to files. However, it should be noted that its efficiency drops drastically as the network latency increases, making it less suitable for larger networks. A second option is Chirp, a distributed file system used in grid computing, for easy user-level export and file access. It combines the elements of both streaming I/O and RPC protocols, and is designed to perform well with large numbers of smaller files, making it highly suitable for our framework. Implementations of Chirp include an internal implementation by Condor, a native protocol implementation used by Parrot [70], as well as a standalone implementation [69] maintained by the University of Notre Dame.

7.3.3 Heterogeneous nodes and OpenCL

While our framework prototype currently supports the Cell BE, it is designed and intended to benefit from multiple architectures concurrently, as a heterogeneous distributed environment. With the GPU becoming increasingly available to general purpose programming, practically every commodity machine becomes a heterogeneous system. Thus, heterogeneity is actually provided on two levels; in the node itself, and in the network. In the current state of the framework, this implies that the programmer has to optimize the user defined functions explicitly for each architecture, to be able to benefit from features like data-level parallelism by SIMD instructions. Furthermore, GPUs are highly parallel processors with hundreds of cores, requiring own frameworks like CUDA to be programmed efficiently. Optimizing and managing code for multiple architectures at once may become a challenging task. We propose investigating OpenCL as a possible solution to this problem.

As described in section 2.5.2, OpenCL provides a common abstraction layer across multiple architectures, allowing code to run on any architecture that supports the framework. In the last couple of years, OpenCL has become more publicly available, with implementations from several industrial actors [47–49]. Although the current implementations for the Cell BE are rather limited, with IBM's XL C/C++ compiler currently being the only compiler and corresponding implementation fully supporting the platform, this is likely to change in the near future with projects like OpenCL PS3. For our framework, OpenCL opens up for the possibility of writing cross-compilable code for multiple architectures. This frees the user from the current architecture specific complexity and constraints, and allows for automatic data-parallel processing where the architecture allows it.

We propose two possibilites for the integration of OpenCL with our framework: the programmer writes the user defined functions in the OpenCL language, cross-compiles the binary for each of the architectures used, and defines the architecture type in the workload. This approach allows for the programmer to maintain fine-grain control over the workloads, as data partitioning and work distribution are managed manually. A second approach is integrating OpenCL as an abstraction layer on top of our framework, where the runtime handles distribution of work-items and work-groups to the heterogeneous nodes automatically. This approach relieves the programmer of work distribution, and lets the user focus on optimizing code in the OpenCL language. However, as discussed in section 2.5.2, OpenCL provides two programming models; a task-parallel model, and a data-parallel model. While the task-parallel model is intended for multi-core machines and the Cell BE, the data-parallel model is intended for architectures such as GPUs. Thus, to perform optimally on all architectures, the framework might need to enforce a hybrid of these.

7.4 Final thoughts

Distributed computing, used as a means for scaling applications, requires a low-level infrastructure that allows for the integration and adaption of applications in all domains. Our framework provides this functionality by allowing the programmer to explicitly offload compute intensive segments of code, and provides an easy-to-use interface to do so. Many problems are in fact embarrassingly parallel, and our framework excels with these. I believe exploiting multiple levels of heterogeneity in distributed environments yields an efficient way of increasing application performance, given that the frameworks are able to mitigate the challenges of parallelization.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have presented a design and a prototype framework that can be used to accelerate applications, by offloading compute intensive parts of a program to a heterogeneous distributed environment. We limited the scope of the thesis to add support for the Cell BE in the framework, and left the extension to additional platforms as further work. The framework was evaluated by implementing two applications, and measuring the throughput gained for each added node. When implementing the applications, we focused on work distribution strategies, and how applications can excel in such environments without large code modifications.

During our investigation, we examined the Cell BE's performance capabilities, and the frameworks targeted for the architecture. While the Cell BE has high potential processing power for direct implementations, we wanted to evaluate the architecture in a distributed environment. To achieve this, we implemented a complete framework for offloading to the Cell BE. The framework consists of two main components: a runtime library, and a scheduler for the Cell BE. The runtime library was designed to transparently handle workload dispatching, connection management, and synchronization events, hiding the underlying complexity of the framework to the user. For the Cell BE, we developed a scheduler, able to queue and execute incoming workloads from the client.

To evaluate the performance of our framework, we implemented an M-JPEG video encoder, where we offloaded the compute intensive parts of the algorithm to the Cell

BEs. We found that the framework provided a good performance increase utilizing a single Cell BE, but had limited scalability with added nodes. We argued that this was caused by the limited number of concurrent workloads the implementation was able to produce, due to post-processing needed after each frame. Combined with potential overhead from network latency, and overhead in the runtime library, this resulted in starvation on the SPEs. With these results in mind, we ported an existing implementation of the AES block cipher, using a work distribution strategy that allowed us to create any number of concurrent workloads. Since all processing was done on the SPEs, we could emphasize runtime overhead by lowering the rate of which workloads were dispatched. However, by adjusting to a higher number of concurrent workloads, we were better able to saturate the SPEs, and the implementation showed a near linear increase in throughput for each node added.

8.2 Further work

During our research, we looked at a number of issues and challenges related to the acceleration of applications in a heterogeneous distributed environment. Unfortunately, we did not find time to answer all questions, and some are left as further work. The Cell BE scheduler and runtime library are both subjects to further optimization, and with regard to the applications, the M-JPEG video encoder shows the most promising potential for improvements.

When implementing the scheduler, we did not have time to test different approaches to programming models on the Cell BE. It would have been interesting to see how an SPE-centric workload scheduler would have performed compared to the PPE-centric model we implemented, where SPEs would fetch the workload, and the PPE could focus entirely on queueing, much like the MARS framework described in section 2.5.1. Such a model would also open up the possibilities of double buffering workloads, which would be a great advantage with the regard to the high latency of DMA transfers. The main obstacle with a SPE-centric scheduler, is that the code sent from the client would need to be dynamically loaded to the LS and executed from the SPE. To our knowledge, libspe2 does not support this functionality, and would have to be done manually. We would also have liked to investigate strategies for loading multiple workloads to each SPE, and then executing them sequentially, as this would most likely reduce scheduling overhead.

Regarding the runtime library, latencies in dispatch overhead and workload synchronization mechanisms should be investigated. The endianess of the client and servers can differ, and the runtime would most likely benefit from a more serialized approach to packing and unpacking workloads. When support for more architectures are added to the framework, the runtime library will also need to have a more efficient load balancer, as workloads will most likely be processed at different rates on each platform. Moreover, it would have been interesting to see how the framework would have performed in conjunction with a distributed file system, as this may help reduce overhead from workload dispatching and data transfers.

The M-JPEG implementation did not have a work distribution strategy that allowed us to dispatch a sufficient rate of workloads, causing starvation on the SPEs when using multiple nodes. Further increases in throughput may be achieved by utilizing the multi-threading capabilities of the client itself, assigning each frame to a dedicated thread. Although this would require a major rewrite of the implementation, it would allow for threads to create workloads simultaneously, thus increasing the dispatch rate.

While both the runtime library and Cell BE scheduler are subjects to further optimizations, the most interesting work would be the extension to support multiple platforms. Although this would require implementing architecture specific schedulers, and most likely modifications to the current workload structure, only minor changes should be required for the runtime library.

8.3 Conclusion

Though heterogeneous multi-core platforms are becoming commonplace, there is no default strategy to reap the potential benefits of these emerging technologies, especially in a distributed environment. Heterogeneous architectures requires per application adaption, and the current frameworks used to exploit the architectures are complex. We have looked at a possible prototype solution for the Cell BE, and achieved increased performance for some applications. To cope with Moore's Law, the number of cores per socket will continue to grow, and cores may become more specialized. Thus, programmers need a way of adapting to the potential performance improvements the architectures have to offer.

Appendix

Attached is a CD-ROM containing the source code, and the benchmark results from the tests. The content can also be found at the following address:

`http://folk.uio.no/martinwa/master/`

Bibliography

[1] David Canter. Chip multi-processing: A method to the madness. 2005. `http://www.realworldtech.com/page.cfm?ArticleID=RWT101405234615`.

[2] IBM Systems and Technology Group. Cell broadband engine programming handbook. pages 45–46, 2008. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D`.

[3] Wikipedia. Amdahl's law — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Amdahl's_law`, 2010. [Online; accessed 02-August-2010].

[4] *MARS - Multicore Application Runtime System*, 2008. `http://ftp.uk.linux.org/pub/linux/Sony-PS3/mars/latest/mars-docs-1.1.5/html/`.

[5] Sony Computer Entertainment Inc. Cell programming primer. 2008. `http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-latest/CellProgrammingPrimer.html`.

[6] Wikipedia. Chroma subsampling — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Chroma_subsampling`, 2010. [Online; accessed 08-September-2010].

[7] Wikipedia. Jpeg — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/JPEG`, 2010. [Online; accessed 08-September-2010].

[8] Wikipedia. Advanced encryption standard — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Advanced_Encryption_Standard`, 2010. [Online; accessed 23-September-2010].

[9] Wikipedia. Block cipher modes of operation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation`, 2010. [Online; accessed 23-September-2010].

[10] Gordon E. Moore. Cramming more components onto integrated circuits. 1965. `ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf`.

[11] William Knight. Two heads are better than one. 51:32–35, 2005.

[12] International Business Machines Corp. The cell project at ibm research. `http://www.research.ibm.com/cell/`.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[15] Tejaswi Redkar. *Windows Azure Platform*. Apress, Berkely, CA, USA, 2010.

[16] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[17] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[18] V. S. Sunderam. Pvm: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[20] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[22] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM J. Res. Dev.*, 53(5):747–758, 2009.

[23] &#142;eljko Vrba, Pal Halvorsen, Carsten Griwodz, Paul Beskow, and Dag Johansen. The nornir run-time system for parallel programs using kahn process networks. In *NPC '09: Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[24] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74, 1974.

[25] NVIDIA. Cuda - compute unified device architecture. `http://www.nvidia.com/object/cuda_home_new.html`.

[26] Intel Corporation. Intel compiler options for sse generation (sse2, sse3, ssse3, sse4.1, sse4.2, avx) and processor-specific optimizations. `http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/`.

[27] Intel Corporation. Intel advanced encryption standard (aes) instructions set rev3. `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/`.

[28] Intel Corporation. Enhanced intel speedstep technology overview. `http://www.intel.com/cd/channel/reseller/asmo-na/eng/203838.htm#overview`.

[29] AMD. Amd cool'n'quiet technology. http://www.amd.com/us/products/technologies/cool-n-quiet/Pages/cool-n-quiet.aspx.

[30] AMD. Amd powernow! technology. http://www.amd.com/us/products/technologies/amd-powernow-technology/pages/amd-powernow-technology.aspx.

[31] 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*, 2005.

[32] David Becker. Playstation 3 chip has split personality. 2005. http://news.cnet.com/PlayStation-3-chip-has-split-personality/2100-1043_3-5566340.html.

[33] Ben Kuchera. Sony answers our questions about the new playstation 3. 2009. http://arstechnica.com/gaming/news/2009/08/sony-answers-our-questions-about-the-new-playstation-3.ars.

[34] Sony Computer Entertainment America LLC. Playstation 3 system software version 3.21. 2010. http://us.playstation.com/support/systemupdates/ps3ps3_321_update1/index.htm.

[35] John Markoff. Military supercomputer sets record, new york times. 2008. http://www.nytimes.com/2008/06/09/technology/09petaflops.html?_r=1.

[36] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Yukio Watanabe Martin Hopkins, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. volume 26, pages 10–24, 2006.

[37] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation. 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf/.

[38] David Krolak. David krolak on the cell broadband engine eib bus. 2005. http://www.ibm.com/developerworks/power/library/pa-expert9/.

[39] International Business Machine Corporation. Software development kit for multicore acceleration - programming tutorial. page 4, 2007.

[40] Abraham Arevalo, Ricardo M. Matinata, Maharaja Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. Programming the cell broadband engine architecture: Examples and best practices. 2008.

[41] Linux for playstation 2 community. http://playstation2-linux.com/projects/ps2linux.

[42] IBM Systems and Technology Group. Xl c/c++ for multicore acceleration for linux. http://www-01.ibm.com/software/awdtools/xlcpp/multicore/.

[43] John L. Gustafson. Reevaluating amdahl's law. pages 92–93, 1995.

[44] IBM Systems and Technology Group. Cell broadband engine programming handbook. pages 699–700, 2008. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D.

[45] IBM Systems and Technology Group. Cell broadband engine programming handbook. pages 402–412, 2008. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D.

[46] Khronos OpenCL Working Group. Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[47] Apple Inc. Opencl: Taking the graphics processor beyond graphics. http://images.apple.com/macosx/technology/docs/OpenCL_TB_brief_20090903.pdf.

[48] AMD. Opencl: The open standard for parallel programming of gpus and multi-core cpus. http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx.

[49] NVIDIA Corporation. Opencl for nvidia. http://www.nvidia.com/object/cuda_opencl_new.html.

[50] IBM Systems and Technology Group. Opencl development kit for linux on power. http://www.alphaworks.ibm.com/tech/opencl.

[51] Robbie McMahon. Opencl on the playstation 3. 2009. http://sites.google.com/site/openclps3/.

[52] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[53] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`.

[54] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[55] Steffen Viken Valvag and Dag Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *NPC '09: Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing*, pages 174–181. IEEE Computer Society, 2009.

[56] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[57] J. M. Merino-Vidal, I. Gelado, and N. Navarro. Evaluation of cell be spu scheduling for multi-programmed systems. 2008. `http://www.ideal.ece.ufl.edu/workshops/wiosca08/paper9.pdf`.

[58] J. Postel. User Datagram Protocol. (768), August 1980.

[59] J. Postel. Transmission Control Protocol. (793), September 1981. Updated by RFCs 1122, 3168.

[60] Eric Hamilton. Jpeg file interchange format. 1992. `www.w3.org/Graphics/JPEG/jfif3.pdf`.

[61] Joint Photographic Experts Group. Jpeg standard. 1992. `http://www.w3.org/Graphics/JPEG/`.

[62] Cipr video test sequences. `http://cipr.rpi.edu/resource/sequences/`.

[63] Specification for the advanced encryption standard (aes). 2001.

[64] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, October 1999. supersedes FIPS 46-2.

[65] Philip J. Erdelsky. Rijndael encryption algorithm. 2002. `http://www.efgh.com/software/rijndael.htm`.

[66] Håvard Espeland. Investigation of parallel programming on heterogeneous multiprocessors. Master's thesis, University of Oslo, 2008.

[67] Stanford University. Folding@home distributed computing. 2010. `http://folding.stanford.edu/English/Main`.

[68] B. Nowicki. NFS: Network File System Protocol specification. (1094), March 1989.

[69] Douglas Thain and Christopher Moretti. Efficient access to many samall files in a filesystem for grid computing. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 243–250, Washington, DC, USA, 2007. IEEE Computer Society.

[70] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. 2003. `http://pages.cs.wisc.edu/~thain/research/parrot/parrot-agm2003.pdf`.