Doctoral Dissertation

Processing Cyclic Multimedia Workloads on Modern Architectures

Håvard Espeland

February 2014

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo

Abstract

Working with modern architectures for high performance applications is increasingly more difficult for programmers as the complexity of both the system architectures and software continue to increase. The level of hand tuning and native adaptations required to achieve high performance comes at the cost of limiting the portability of the software. For instance, we show that a compute intensive DCT algorithm performs better on graphic processors than the best algorithm for x86. In particular, limited portability is true for cyclic multimedia workloads, a set of programs that run continuously with strict requirements for high performance and low latency. An example of a typical multimedia workload is a pipeline of many small image processing algorithms working in tandem to complete a particular task. The input can be videos from one or more live cameras, and the output is a set of video frames with elements from several of the source videos, for example as stitched panorama frames or 3D warped video. Such a setup runs continuously and potentially needs to adapt to various degrees of changes in the setup without interruptions or downtime.

To reach the performance goal required by multimedia pipelines, modern, heterogeneous architectures are considered instead of the traditional symmetric multi-processing architectures. We also investigate variations between recent microarchitectures of symmetric processors to identify differences that a low-level scheduler must take into account. Further, since multimedia workloads often need to adapt to various external conditions, e.g., adding another participant to a video conference, we also investigate elastic and portable processing of multimedia workloads. To do this, we propose a framework design and language, which we call P2G. In the age of Big Data, this idea differs from the typical frameworks used for distributed processing, such as MapReduce and Dryad, in that it is designed for continuous operation instead of batch processing of large workloads. We emphasize heterogeneous support and expose parallel opportunities in workloads in a way that is easy to target since it is similar to sequential execution with multidimensional arrays. The framework ideas are implemented as a prototype and released as an open source platform for further experimentation and evaluation.

Acknowledgements

The PhD experience has been a roller coaster ride from the first day as a candidate until finishing my thesis. There have been periods of incredible stressful deadlines, but most days have filled with heaps of fun exploring systems, problems and doing research with the best possible colleagues at Simula. Most of all, I would like to thank my advisors Pål Halvorsen and Carsten Griwodz for their deep insights and valuable feedback over the years. For great scientific discussions at Simula Research Laboratory, I would like to especially thank Paul Beskow, Håkon Stensland, Andreas Petlund and Zeljko Vrba. Without them, many ideas would not have come to a fruitful realization. Going to work is a delight at Simula. In addition to the already mentioned group, I would like to thank Ragnhild Eg, Preben Olsen and Tomas Kupka together with the rest of MPG for always making Simula a great place to be. Thanks to my girlfriend, Cathrine Skorpen Nygaard, whom I met during the last phase of writing this dissertation. Meeting you is the best thing that has ever happened to me! Finally, thanks to my family and friends. You have always supported me, no matter what I do.

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction

1.1 Background

Multimedia processing is a highly relevant topic with regard to today's massive production and consumption of multimedia content. For example, Youtube reported[1] 72 hours of videos uploaded every minute in 2013, and over 4 billion hours of video watched every month. Similarly in the mobile domain, 51% of the traffic was mobile video in 2012 [2]. In order to utilize the content in a useful manner, it must be processed for the intended use. One such example can be video broadcasted to millions of users on streaming sites such as Youtube. Raw video has very large storage requirement, and is as such very inconvenient to transfer to end users. Instead, video compression algorithms are used to reduce the storage requirements while preserving most of the content's fidelity. Other examples of multimedia processing algorithms commonly used on video are color grading, feature extraction, noise reduction, or just about any other imaginable operation that can be performed on a video clip. Multimedia processing is of course not limited to video; any content such as sound, 3D or sensory data is applicable. There are several definitions for multimedia with answers depending on who you ask. Although there is no strict definition, a reasonable explanation of the term was defined by Steinmetz and Nahrstedt [3]:

"A multimedia system is characterized by computer-controlled, integrated production, manipulation, presentation, storage and communication of independent information, which is encoded at least through a continuous (time-dependent) and a discrete (time-independent) medium."

It is our goal to provide support for continuous and complex multimedia pipelines, allow-

[1]http://www.youtube.com/yt/press/statistics.html, retrieved April 2013.

ing combinations of individual algorithms to work together in tandem in a real time system, such as free view rendering, which enables virtual cameras to create a synthetic view in-between real shootage and requires large computational resources. Complex pipelines have many inter-dependencies between individual components, and exploiting parallel opportunities is far from trivial.

Processing multimedia by itself is not a new topic, evident by the founding of the ACM Multimedia conference back in 1993 by the multimedia science community, which is still the leading conference in the field. Digital multimedia processing is even older than this, with notable main-stream use of digital special effects in film and TV such as Tron (1982) and The Abyss (1988). Even long before ACM Multimedia, the science community established the SIGGRAPH conference in 1974, which still is the premier conference for publishing research in computer graphics. Hence, a lot of research has been done in the field of multimedia processing to this date. To build upon and extend the state of the art in this thesis, we focus on how the specific characteristics of multimedia workloads applies to modern computer architectures and their implementations. In particular, we look at a set of multimedia workloads that are continuous, iterative, time dependent and highly data parallel. This is not a catch-all, but encompasses common problems that require significant computational resources, such as video encoding, feature extraction, 3D reconstruction and many more multimedia problems.

1.1.1 Modern architectures

Computer architectures are changing rapidly, with market leaders launching new processors every year. The basic structure has been stable for a very long time, but seen from the consumer side, a significant shift could be observed in the mid 2000s; consumer computers came fitted with dual core processors such as the Intel Pentium D [4], something that was previously only available on high end systems and servers before that. For example, Intel provided the Xeon line of products that allowed several identical processors to be connected by a bus and work together as a multicore system. After the Pentium D, chip multiprocessors became a reasonably priced commodity, that is, several cores on a single chip and has since then been standard in the consumer marked. The traditional workloads for multicore machines had been compute bound programs, and with availability of consumer grade multicore, all programs had to be targeted for these type of processors. At the same time, graphics processor units (GPUs) became fully programmable, previously mainly used for rendering game graphics, and could now be used for general purpose computing. With Nvidia launching CUDA [5] in 2007, a revolution in low cost, high performance computing was apparent. The graphics processors were also pioneering high performance computing, not by being first, but by being available at moderate prices to the masses. The GPU architecture differs from symmetric processors such as the sole Pentium D in that processing elements have different capabilities than the main processor, and the whole system can be seen as a heterogeneous system. Similarly, the Playstation 3 was launched in 2006 and is a great example of what can be achieved with a heterogeneous architecture as we will see in later chapters. Even though the consumers first saw these machines in the mid 2000s, heterogeneous architectures were available long before that in various consumer and specialized products. One example is the specialized network processor IXP1200 [6] by Intel, which was launched in 1999, and provided high performance and flexible hardware for network processing in contrast to the traditional approach of implementing network functions in hardware. The flexibility allowed rapid prototyping of network equipment, otherwise impossible to implement in software, but also a relatively low cost platform for deploying if large product volumes were unfeasible.

Obviously, there are significant challenges in the switch from singlecore to multicore, first of all that legacy programs written without a parallel design would get very little benefit from it at all, other than what the operating system can provide. Synchronization issues and parallel algorithms became intrinsic parts of a program's design, which required restructuring or a complete rewrite of existing programs. Further, handling several types of memory and caches puts an additional burden on the programmers writing programs on these architectures. It may be argued that the switch from single to multi core shifted the burden of scaling the system from processor designers to software writers, although the reason for the shift was rather technology driven. Processor designers were unable to increase the clock frequency of a single core further, and to keep consumers satisfied and CPU vendor's current with Moore's law, they put multiple independent cores on a single chip die. Some architectures such as GPUs have even exceeded Moore's predictions, putting far more transistors on a chip die than traditional x86 processors. Our experience with using these architectures for multimedia processing is presented in chapter 2.

1.1.2 Multimedia workloads

Multimedia workloads is a broad term, and to limit the scope of this thesis, we look at workloads that are iterative, time sensitive and revolve around video processing. Video processing is typically a very data intensive operation, requiring calculations for each individual pixel in a video frame. One common operation is video encoding, reducing storage and bandwidth requirements of video in exchange for decoding complexity. The most prominent video codec standard today is H.264 [7], defined by ISO and the International Telecommunication Union (ITU), and used for video displayed on everything from mobile phones to traditional linear TV broadcasting. Being so popular also means that much research has been done to optimize its performance, and in the setting of live video, dedicated hardware encoders are commonly used. Dedicated encoding hardware solves the encoding problem in production settings in a convenient way, with strict guarantees of meeting deadlines and with low power requirements. Similar hardware implementations exist for many multimedia processing workloads used in a production setting. The main limitation of hardware implementations is the lack of flexibility, prohibiting changes both in workloads and input/output data. In production environments, this static nature may be acceptable, and the advantages of using tailored hardware is huge, including low power consumption, high performance, guaranteed real time capabilities and so on. Designing and fabricating a hardware chip is an expensive and time demanding process and is typically only done for standardized components such as video codecs and other commonly used filters. Multimedia workloads developed in software retain flexibility, allowing easy modifications, updates and replacements in comparison to hardware implementations, and in this work, we will focus on software implementations of multimedia workloads, running on modern architectures.

In particular, this thesis targets workloads that are data intensive and applicable to video processing. When we discuss multimedia workloads in this thesis, we mean pipelines of interdependent filters connected in such a way that in tandem they solve a particular multimedia task or problem. As an example of a multimedia workloads, consider the image stitching pipeline in OpenCV [8] that combines several images into one large panorama, emulating a wide angel shot covering all of the input images. The pipeline has many distinct multimedia operations connected with complex dependencies and comprises a series of filters for estimating parameters using invariant features, optimizing and correcting the inputs images, and finally compositing the results into a panorama image. An overview of this pipeline can be seen in figure 1.1. Implementing the pipeline to execute sequentially is straight forward and the challenge arises when parallel execution is desired. Because of the dependencies, some filters must be performed before others, and ideally, one would want to start the next filter operation as soon as all dependencies are satisfied from previous operations, i.e., before a filter has been run to completion on all data. Another problem is the scalability of input data, that is, adding more input images or higher resolution yields more computational requirements resulting in more work to be performed by the filters. Further, complex filters with non-linear computation time may skew the proportional amount of work that will have to be performed in one filter compared to the rest.

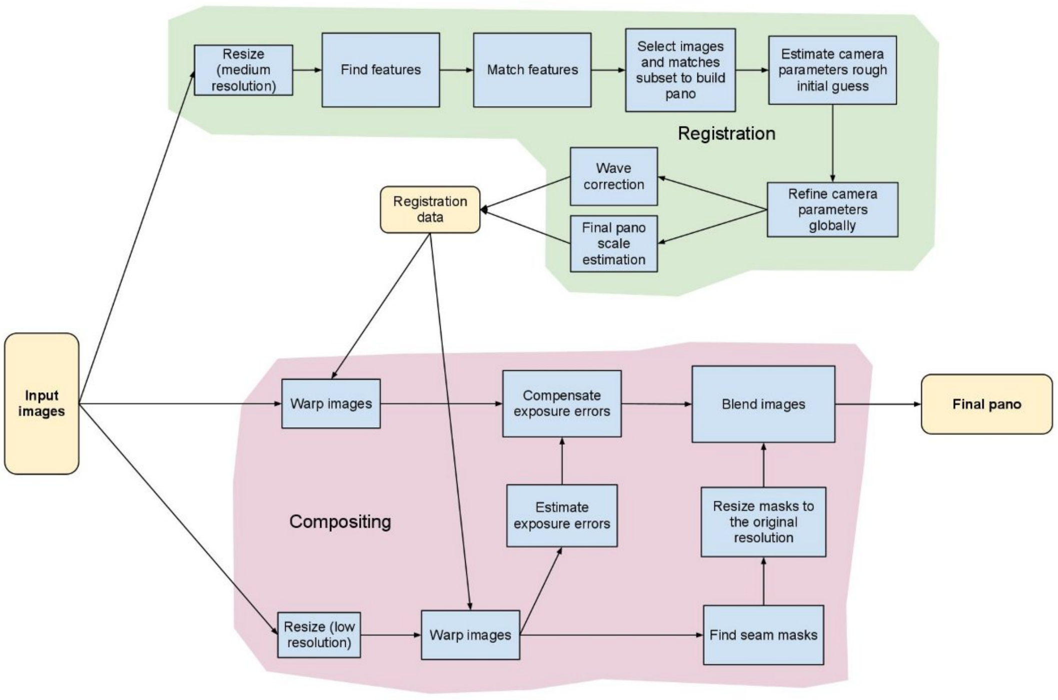

Resize (medium resolution)
Find features
Match features
Select images and matches subset to build pano
Estimate camera parameters rough initial guess
Registration
Wave correction
Registration data
Refine camera parameters globally
Final pano scale estimation
Input images
Warp images
Compensate exposure errors
Blend images
Final pano
Compositing
Estimate exposure errors
Resize masks to the original resolution
Resize (low resolution)
Warp images
Find seam masks

Figure 1.1: Image stitching pipeline example available in OpenCV.

1.1.3 Elastic multimedia processing

Elasticity in the realm of cloud computing refers to dynamic scalability, not to be confused with elasticity in the networking field. Buyya et. al. [9] defines elastic computing as a system able to provision additional resources when an application load increases, and release them when the load decreases. This is a very desirable feature for processing frameworks and indeed for multimedia processing.

As a motivation for our work, we looked at a specific multimedia processing scenario and how to best implement this using existing approaches. The basic idea is one or more live video feeds providing a video source, followed by a black box doing a set of video processing filters, and finally a sink providing the output to the end user. In principle, such a system is straightforward to implement with each task or block of data neatly divided among the available CPU cores with a static schedule processing a continuous flow of multimedia content. The baseline is appropriate for a fixed system, but we can imagine several scenarios where such a setup would require significant changes or even a full rewrite of the system to implement the changes:

- Different inputs, e.g., higher resolution video or more simultaneous input streams could result in exhausting the resources of a static system. This is in contrast to an elastic system that can adapt to the requirements.
- Varying resource availability caused by resource sharing in the OS or by virtualization.
- Deadline misses caused by too high processing requirements at times, in a scenario were the complexity can be tuned (such as in a video encoder).
- If the problem is too resource intensive to be processed on a single machine, several nodes can contribute in a distributed system.

Designing and implementing an elastic system that can handle varying input data, heterogeneous computational resources and even varying processing goals is a much more challenging problem than a fixed processing pipeline. The demand for elasticity may not be obvious until the demand for change arises, e.g., by requiring more input streams than the system was designed for, or adding a new filter to the pipeline that results in a system not scaling within the acceptable parameters. In the best case, better hardware can be added to solve the new problem, and in the worst, a complete rewrite or port to a different platform has to be performed. Depending on the requirements, providing dynamic scalability to the system may help with planning for unforeseen usage. Further, elasticity allows the available resources to change during runtime, enabling resource sharing and virtualization.

The term elastic computing is often used in combination with cloud computing, although they are not strictly synonymous. Elasticity as such is a feature of cloud computing, with the term coined by Amazon as part of their Elastic Cloud Computing (EC2) platform [10], providing dynamic scalability and utilization of resources at hand. Cloud computing is used in many contexts including internet clouds such as Amazon's EC2 or in private clouds, for instance Hadoop [11] clusters. Although elasticity is a basic requirement for cloud computing, we are not focusing on cloud computing in this thesis.

Many systems exist today for elastic execution of workloads, with the MapReduce paradigm [12] being most prominent, but also several others that we will discuss later in this thesis. What we find in common for most of them is that they are designed for *batch processing* instead of continuous and iterative operations. Even though systems for continuous operations that are suitable for multimedia workloads exist, e.g., StreamIT [13], they are not designed to provide solutions for the scenarios presented above. This does not imply that they cannot be used for this type of work, but since they are not intrinsically designed for elastic execution of multimedia workloads, the programmer basically has to shoehorn the workloads into fitting their model, which leads to problems both in flexibility and performance. We will go further into this topic when we look at elastic execution in chapter 3.

1.2 Problem Statement

Modern multicore processors are great for processing multimedia workloads. However, they require programs that are specifically optimized for the particular architecture. This inhibits portability and scalability by requiring large refactoring jobs or even complete rewrites when the programs are highly optimized and written natively for them. In contrast to this, there is elastic execution design for writing once, running everywhere. Naively, this can be done in a virtualized or interpreted language, but this does not provide high performance and efficient execution of the workloads. Many systems for elastic execution exist, but they are not tailored for multimedia workloads, and most of them are designed for batch (single iteration, non realtime) processing. Further, fluctuating resources and varying workloads provide a constantly changing working environment, hence a static design is insufficient. To explore this area of computing, we have worked under the following problem statement:

Can a pipeline of cyclic multimedia workloads be executed by an elastic framework on modern microarchitectures in a portable and efficient manner?

The problem stated in this thesis encompasses how to support multimedia processing on modern architectures from a system's perspective. In particular, our focus is on elastic computing and how to take advantage of computational resources in a dynamic manner adapting to the system architecture, resource availability and application requirements. To approach this, we have split the problem into three sub-problems, presented in chapters 2 and 3:

1. Identify key architectual considerations on modern microarchitectures required to process cyclic multimedia workloads efficiently;
2. Based on these key observations, design a programming model and framework to expose

parallel opportunities in the workloads for efficient processing, providing elastic execution capabilities;

3. Implement a prototype and evaluate the prototype's feasibility to solve the problem and its achieved performance.

1.3 Limitations

To limit the scope of this thesis to a feasible project within the frame of a thesis, we have focused this work on cyclic (or continuous) multimedia workloads that are data intensive and have large parallel opportunities; even though scheduling is an important aspect of efficient execution, we have not considered scheduling strategies in the framework presented in later chapters and left this as further work. Nor have we had the chance to pursue distributed execution, which falls naturally within this scope. Since both are important for elastic execution, we have designed our prototypes and experimental frameworks with these aspects in mind, allowing future work to expand knowledge within these areas.

1.4 Research Method

The discipline of computing, computer science is divided into three major paradigms as defined by the ACM Education Board [14] in 1989. Each of them have roots in different areas of science, although all can be applied to computing. The board states that all paradigms are so intricately intertwined that it is irrational to say that any one is fundamental in the discipline. The three paradigms or approaches to computer science are:

- The *theory* paradigm is rooted in mathematics and comprises defining objects of study, hypothesizing relationships among them, determining whether said relationships are true and interpreting the results. The paradigm is concerned with the ability to describe and prove relationships among objects.

- The *abstraction* paradigm is rooted in experimental research and consists of four stages. In this paradigm, the scientist forms a hypothesis, constructs a model and makes a prediction, before designing an experiment and collects data. The paradigm is concerned with the ability to use those relationships to make predictions that can be compared with the world.

- The *design* paradigm is rooted in engineering and involves building a system or device to solve a given problem. The scientist states requirements and specifications, followed by design and implementation of said system. Finally, the system is tested and the steps can be iterated if the tests reveal that the system does not meet the requirements. The paradigm is concerned with the ability to implement specific instances of the relationships and use them to perform useful actions.

The ACM board further states that applied mathematicians, computational scientists and design engineers generally do not have interchangeable skills. All three paradigms are used in the discipline of multimedia research, depending on the problem at hand and what one expects to achieve.

For investigating how to perform efficient multimedia processing on modern architectures, we first consider the *theory* paradigm for our research. As the *theory* paradigm is rooted in mathematics, it requires very precise definition and modeling to correctly represent complex dependencies found both in the software of the multimedia workloads and execution system and the computer architecture itself. The paradigm is a better fit for isolated problems where objects of study are limited and all objects are understood. Since we are dealing with what are essentially black boxes made by CPU vendors, and the complexity of these boxes is very high, we reject this paradigm for our work. The second paradigm, *abstraction*, is heavily used in our field, typically in combination with simulators. By modelling both the hardware architecture and software in a simulation, scientists can accurately capture the behaviour and inspect states otherwise not available in a prototype implementation. The downside with simulators is that they are only as good as the model used, and in our particular case, there are many black boxes (e.g., hardware), which not only are very hard to model, but also undocumented and thus impossible to model accurately. As such, the hardware model has to be a simplification of the real system behaviour, and the trade-offs done in the model may or may not be sufficient to validate the hypothesis. The problem is that one only validates the hypothesis for the model, and not the full system. Hence, it is impossible to know if the simulated behaviour is congruent with the real system without experiments. This does not imply that simulations are useless, but rather that they should be used in combination with prototype implementations when the model is a simplification of the real world. Finally, we consider the *design* paradigm, where a prototype is built and experimental data are collected from real working systems. This paradigm allows the scientist to specify requirements and build prototypes based on them. The prototype is thus evaluated with experiments, and based on them, the prototype can be updated with several iterations. This can be done in combination with simulations to verify the results, or as stand alone experiments. The downside with the *design* paradigm is the significant engineering effort required to build working prototypes that itself is not scientific work, but still essential to produce final results.

In the problem statement, we said that we investigate modern microarchitectures, and even though some documentation on them exists, they are still black boxes, created by vendors in their labs with undocumented as well as unexpected behaviour hidden both in silicon and in microcode. The undisclosed nature of this design inhibits detailed models to be made in addition to their highly complex structure, and a model based simulation would require a high level of simplification. Simplifications do not imply that the model is wrong, but deciding if the simulation results match reality is up for interpretation. Consequently, we decided to implement real working prototypes using the *design* paradigm for the work in this thesis. It requires more engineering work, but is not limited by inaccurate or simplified models as basis for simulations of the systems.

1.5 Main Contributions

The research question posed in section 1.2, states a challenge of efficient multimedia processing on modern architectures. We have addressed this problem on several levels, from low level architecture specific optimizations to a high level programming model support. The contributions of this thesis have been presented at a number of peer reviewed conference proceedings listed in full in chapter 4. In addition to the contributions listed below, we have written a number of other conference and journal papers related to multimedia processing that are not included here to limit the scope of this thesis. A summary of the included contributions follows:

- The Intel IXP architecture was used as a platform for prototype experiments involving video protocol translation and provided architectual and network protocol insights for a video streaming workload. The prototype allowed efficient processing on modern hardware, validating a network protocol optimization while benefiting from heterogeneous processing using a programmable network processor.

- Scalability and architectual considerations for video encoding were evaluated and compared with the same problems both on IBM's Cell Broadband Engine and NVIDIA's GPUs. Intel's x86 was used as baseline and the suitability of different algorithms in addition to specific memory considerations were quantified and evaluated. Among the lessons learned, we saw how algorithms performing better on one architecture were completely outperformed on others, emphasizing the importance of allowing different algorithms to be used interchangeably and dynamically to support portable multimedia pipelines to perform efficiently in concert.

- Symmetric multicore processors from different vendors allow a convenient and portable programming model, but although the interface between implementations is identical, we show that the scheduling order of threads and data access patterns severely impacts performance and the best performing low level schedules depend heavily on the data set, affinity and micro architecture used. Hence, the best schedule for Intel's CPUs may not be the best performing schedule for AMD's latest processors.
- Using the knowledge of structuring and optimizing multimedia workloads for heterogeneous systems, we propose P2G, comprising a language and framework that exposes parallel opportunities to a runtime system and allows elastic execution of cyclic multimedia workloads in an convenient manner. The runtime system allows several levels of optimizations to be performed at compile time, runtime or just-in-time.
- A prototype of the *P2G* framework is implemented together with several multimedia workloads. Finally, the *P2G* framework is evaluated with micro-benchmark experiments, demonstrating that the framework is useful for multimedia workloads.

Even though there are many opportunities for further work in several of these areas, we aim to show the reader some of the considerations that are necessary and opportunities available to process cyclic multimedia workloads on modern multicore architectures in an efficient manner. The research area, as defined by the problem statement, is very wide and a complete coverage within a thesis would be impossible. Still, we think our findings ranging from low level issues and operating system support to high level programming model constitute a significant contribution to the field and state of the art.

1.6 Outline

The rest of the thesis is organized in two parts. Part I gives an introduction to the research papers and puts them into context, while Part II includes the papers in full. The first part is organized as follows: In chapter 2, we consider three case studies of using multimedia workloads on modern architectures, and draw conclusions on how the workloads have to be structured. We also discuss low-level scheduling implications imperative to efficient processing on some modern architectures. Further, in chapter 3, we use the knowledge gained to design a parallel processing framework (P2G), comprising a high level model and runtime that supports processing of multimedia workloads on heterogeneous systems. We provide a set of multimedia workloads written for the P2G language and present a prototype implementation, covering a subset of the features of the P2G system. The performance is evaluated in a set of benchmarks and discussed in detail. This is followed by a overview of the research papers included in the thesis in chapter 4. Finally, a conclusion to the thesis is given in chapter 5.

Chapter 2

Architectural considerations

In this chapter, we look at what architecture differences to consider when running multimedia workloads on modern multiprocessors. We classify and emphasize differences between three groups of architectures, and present a case study on using them for multimedia processing. Since the architectures differ significantly, we have used these case studies as motivation for how to design a language and runtime in later chapters for portable and elastic multimedia processing on heterogeneous architectures.

The main differences between the various architectures presented are the amount of computational resources (ALUs, FPUs, etc.) and memory layout (memory types, caches, latency and bandwidth). Multimedia workloads are typically heavily data parallel with relatively few inter-dependencies between small pieces of data, making heterogeneous architectures with many cores a perfect match. In their memory layout, we see large variations between processors presenting both challenges and opportunities for our workloads. In this regard, we have divided common architectures in three classes based on their memory characteristics and present prominent examples together with case studies about using this particular types of architecture for multimedia workloads.

2.1 Symmetric shared memory architectures

2.1.1 Overview

The most common multicore processor architecture today is symmetric shared memory (SMP). Examples include Intel x86, ARM Cortex and IBM PowerPC, each with many variations and manifacturers. With SMPs, all cores are equal (in theory) and has direct access any piece of data, regardless if it resides in system memory or another core's cache. SMPs are easy to program as most of the complexity is buried in hardware. An SMP architecture enables any piece of code to run on any core, but code and data placement has performance implications, as we will show in section 2.1.2.

Even though the cores are supposed to be identical, in modern implementations with many cores on a single package, some cores may share resources or have varying degree of connectivity to shared resources such as system memory. For example, the system memory can be segmented in such a way that only part of the memory is directly connected to each core, and access to other parts of the system memory requires traversal via other cores by a network of interconnects. Another example is the AMD Bulldozer architecture where two cores are packaged in a single module. Two modules share a single floating point unit and are connected by a shared L2 cache. Depending on the data flow and computational requirements of the workloads, it is reasonable to assume that even though the architecture is symmetrical, placement and partitioning of code and data can have significant performance implications.

2.1.2 Case study: Cyclic multimedia workloads on x86

There is an ever-growing demand for multimedia processing resources, and the trend is to move large parallel and distributed computations to huge data centers or cloud computing, which typically comprise a large number of commodity x86 machines. As an example, Internet users uploaded one hour of video to YouTube every second in January 2012 [15] and each of these is encoded using several filters. The filters are arranged as vertices in dependency graphs, where each filter implements a particular algorithm representing one stage of a processing pipeline. In this section, we focus on utilizing available processing resources in the best possible manner for time-dependent cyclic workloads. This kind of workload is typical for multimedia processing, where large amounts of data are processed through various filters organized in a data-intensive processing pipeline (or graph).

Several different architectures exist and as we will see in later sections, require very different architectural adaptations to fully utilize them for multimedia processing. Now, with the prevalance of many-core x86 processors, one may assume that as long as your program is optimized for a particular architecture, it will also perform well on similar microarchitectures (e.g., other vendors or generations). In this section we test this assumption for some cyclic multimedia workloads. For an automatic workload scheduler to be able to get the best performance, it has to take local knowledge into account. We have therefore experimented with cyclic workloads on some prevalent multicore microarchitectures, and structured the execution of them using different configurations to see how it affects performance. The results from this work were first presented in an article at IEEE SRMPDS 2012 [16].

Background

Performance implications for scheduling decisions on various microarchitectures have been studied for a long time. Moreover, since the architectures are constantly being revised, scheduling decisions that were preferred in the past can harm performance on later generations or competing microarchitectures. We look at implications for four current microarchitectures and how the order of execution affects the performance. Of recent work on x86, Kazempour et al. [17] looked at performance implications for cache affinity on Clowertown generation processors. It differs from the latest microarchitectures by having only two layers of cache and only the 32 KB L1 D-cache is private to the cores on a chip multiprocessor (CMP). In their results, affinity had no effect on performance on a single chip since reloading L1 is cheap. When using multiple CMPs, on the other hand, they found significant differences meriting affinity awareness. With the latest generation CMPs having significantly larger private caches (e.g., 256 KB on Intel Sandy Bridge, 2 MB on AMD Bulldozer), we can expect different behavior than on Clowertown. In terms of schedulers that take advantage of cache affinity, several improvements have been proposed to the Work Stealing model. Acar et al. [18] have shown that the randomized stealing of tasks is cache unfriendly and suggest a model that prefers stealing tasks where the worker thread has affinity with that task.

In this case-study, we evaluate how a processing pipeline of real-world multimedia filters runs on state-of-the-art processors. We expected performance differences between different architectures, but were surprised to find large differences between microarchitectures even within the same family of processors (x86) making it is hard to make scheduling decisions for efficient execution. For example, our experiments show that looking into cache usage and task dependencies can yield large performance gains. There are also significant differences in the configuration of the processing stages, e.g., when changing the amount of rotation in an image, giving completely different resource requirements. Based on these observations, it is clear that the low-level scheduling should not only depend on the processing pipeline with the dependencies between tasks, but also the specific micro-architecture and the size of the caches. We discuss why scheduling approaches such as standard work stealing models do not result in an optimal performance, and we try to give some insights that a future scheduler should follow.

Design and Implementation

Inspired by prevalent execution systems such as StreamIT [13] and Cilk [19], we look at ways to execute data-intensive streaming media workloads better and examine how the execution order affects performance. We also investigate how these behave on different processors. Because we are processing continuous data streams, we are not able to exploit task parallelism, e.g., by processing independent frames of a video in parallel; and therefore seek to parallelize within the data domain. A scenario with embarrassingly parallel workloads, i.e., where every data element can be processed independently and without synchronization, is video stream processing. We believe that processing a continuous flow of video frames is a reasonable example for several embarrassingly parallel data bound workloads. In this work, we study two approaches for executing workloads; they differ by what order the data is accessed to induce effects in cache behaviour, and as a result the system's performance.

Our **sequential approach** is a straight-forward execution structure where a number of filters are processed sequentially (in a pipeline), each frame is divided spatially and processed independently by one or more threads. In each pipeline stage, the worker threads are created, started, and eventually joined when finished. This would be the natural way of structuring the execution of a multithreaded media pipeline in a standalone application. Such processing pattern has natural barriers between each stage of the pipeline. For a execution system such as StreamIT, Cilk, Multicore Haskell [20], Threading Building Blocks [21] and others that use a work stealing model [22], the pattern of execution is similar to the sequential approach, but this depends very much on how work units are assigned to worker threads, the workloads that are running simultaneously and scheduling order. Nevertheless, sequential execution is the baseline of our evaluation since it processes each filter in the pipeline in a *natural* order.

As an alternative approach, we propose using **backward dependencies** (BD) to execute workloads. This approach only considers the last stage of the pipeline for a spatial division among threads and such avoids the barriers between each pipeline stage. Furthermore, for each pixel in the output frame, the filter backtracks dependencies and acquires the necessary pixel(s) from the previous filters. This is done recursively and does not require intermediate pixels to be stored to memory. Figure 2.1 illustrates dependencies between three frames connected by two filters. The pixels in frame 2 are generated when needed by filter B using filter A. The BD approach has the advantage of only computing the pixels that are needed by subsequent filters. The drawback, however, is that intermediate data must be re-computed if they are accessed multiple times because intermediate results are not stored. These re-computations can be mitigated by using the

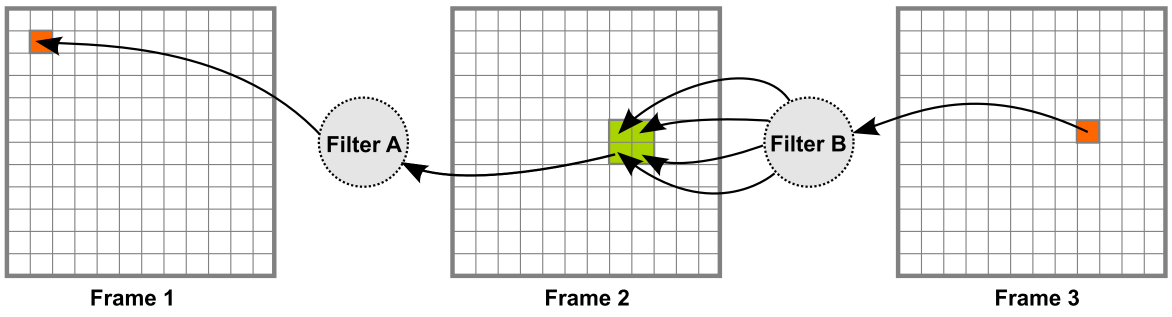


Figure 2.1: Backward dependencies (BD) example for two filters processing frames in a pipeline. The arrows indicate what pixels are required from the previous frame to generate the current pixel. Only one pixel's dependencies per frame are illustrated, other pixels have similar dependencies.

intermediate frames as buffer caches between filters, although the overhead of managing and checking this buffer cache can be large, which we see later in this section.

The backward route for the BD access pattern is deterministic and will produce the same result as the sequential approach. The approach has three steps:

1. Select a pixel for computation in the final frame (output image). Pixels can be computed in parallel, side effect free, on multiple processors.
2. Determine which pixels in intermediate frames need to be computed to produce the selected pixel and compile a list of pixel dependencies.
3. For each pixel dependency, recursively repeat from step 1 to produce intermediate results until the dependency refers to the input frame.

The BD approach has the effect that it does not store intermediate results in memory, resulting in less memory writes. The drawback is that if intermediate results are needed more than once, they must be recalculated. As such, we expect the BD approach to perform better than sequential when regenerating intermediate results is cheaper than storing and retrieving them from memory. We can also mitigate this by committing intermediate results to memory, and use this as a buffer cache when checking dependencies. Managing the cache also requires resources and we investigate if this approach is beneficial later on.

The different approaches incur varying cache access patterns. Depending on memory access patterns, execution structure and CPU microarchitecture, we expect the performance to change. The sequential approach accesses the buffers within a filter in sequential order, and the prefetch unit is thus able to predict the access pattern. A drawback of this approach is that data moved between filters do not necessarily reside in the cache of the core using the data recently. First, this includes data whose cache line has been evicted and written back to a cache level with increased access time or memory. This may happen because the data size processed by a filter is larger than the amount of cache available, forcing write-back. Other reasons include context switches and shared caches. Second, output from a previous filter may not have been generated on the same core as the one that accesses the data, resulting in accesses to dirty cache lines on other cores. Given the spatial division of a frame within a filter, this sounds easy to avoid, but an area of input to a filter may result in output to a different spatial area, which the processor's prefetcher may not be able to predict. Thus, re-using the same core for the same part of a frame for multiple filters in a pipeline only increases cache locality for filters whose source pixels map spatially to the destination pixels. To increase cache locality between filters, we also evaluate the BD approach, where data is accessed in the order needed to satisfy dependencies for the next filter in the pipeline. This ensures that pixels are accessed in a manner where data in between filters are likely to reside in the core's cache. That is to say, if the access pattern is not random, one can expect BD execution to always access spatially close memory addresses.

Experimental Setup

To evaluate the approaches and find what apporach performs best in a low-level scheduler, we built an experimental framework supporting the proposed execution structures and wrote a set of image processing filters as a case study working on real-world data. The filters were arranged in different pipelines to induce behaviour differences that can impact performance.

For all experiments we measure computation time exclusively, i.e., the wall clock time of the parallel execution, excluding I/O and setup time. We use this instead of CPU time to get measurements on how good performance is actually possible, since having used only half of the CPU time available does not mean that only half of the CPU's resources are utilized (cache, memory bandwidth, etc.). Also, by not counting I/O time, we remove a constant factor present in all execution structures, which better captures the results of this study. Each experiment is run 30 times, and the reported computation time is the average running time. All filters use 32-bit float computations, and we have strived to reduce overhead during execution, such as removing function calls in the inner-loop and redundant calculations. Our data set for experiments consists of the two standard video test sequences*foreman* and *tractor* [23]. The former has a 352x288 pixel (CIF, 4:2:0) resolution with 300 frames of YUV data, the latter has 1920x1080 pixels (HD, 4:2:0) with 690 frames of YUV data.

Microarchitecture	CPU	Cores (SMT)	Private Cache	Shared Cache
Nehalem	Intel i5-750	4	64 kB L1 256 kB L2	8 MB L3
Sandy Bridge	Intel i7-2600	4 (8)	64 kB L1, 256 kB L2	8 MB L3
Sandy Bridge-E	Intel i7-3930K	6 (12)	64 kB L1, 256 kB L2	12 MB L3
Bulldozer	AMD FX 8192	8 (4x2)	64 kB L1[1], 2 MB L2[1]	8 MB L3

[1] Shared between two modules each having a separate integer unit while sharing an FPU.

Table 2.1: CPU microarchitectures used in experiments.

The experiments have been performed on a set of modern microarchitectures as listed in table 2.1. The CPUs have 4 to 8 cores and different cache hierarchies: While the Nehalem has an L3 cache shared by all cores, which operates at a different clock frequency than the cores and is called the *uncore*, the Sandy Bridge(-E) has a slice of the L3 cache assigned to each core and accesses the other parts using a ring interconnect running at core speed. Our version of the Bulldozer architecture consists of four modules, each containing two cores. On each module, L1 and L2 are shared between the two cores with separate integer units, but with a single shared FPU.

We have developed a set of image processing filters for evaluating the execution structures. The filters are all data-intensive, but vary in terms of the number of input pixels needed to produce a single output pixel. The filters are later combined in various configurations referred to as pipelines. A short summary of the filters and their dependencies is given in table 2.2.

The filters are combined in various configurations into pipelines (as in figure 2.1). The evaluated pipelines are listed in table 2.3. The pipelines combine the filters in manners that induce different amounts of work per pixel, as seen in the table. For some filters, not all intermediate data are used by later filters and are unnecessary to produce the final output. The BD approach will not produce these, e.g., a crop filter as seen in pipeline B will not require earlier filters to produce unused data. Another aspect that we expect to influence the results is cache prefetching. This means that by having filters that emit data in a different spatial position relative to its input, e.g., the rotation filter, we expect the prefetcher to contend fetching the relevant data.

Scalability

The pipelines are embarrassingly parallel, i.e., no locking is needed and they should therefore scale linearly with the number of cores used. For example, using four cores is expected to yield a

Blur convolves the source frame with a Gaussian kernel to remove pixel noise.

Sobel X and Y are two filters that also convolve the input frame, but these filters apply the Sobel operator used in edge detection.

Sobel Magnitude calculates the approximate gradient magnitude using the results from Sobel X and Sobel Y.

Threshold unset every pixel value in a frame below or above a specified threshold.

Undistort removes barrel distortion in frames captured with wide-angle lenses. Uses bilinear interpolation to create a smooth end result.

Crop removes 20% of the source frame's height and width, e.g., a frame with a 1920x1080 resolution would be reduced to 1536x864.

Rotation rotates the source frame by a specified number of degrees. Bilinear interpolation is used to interpolate subpixel coordinates.

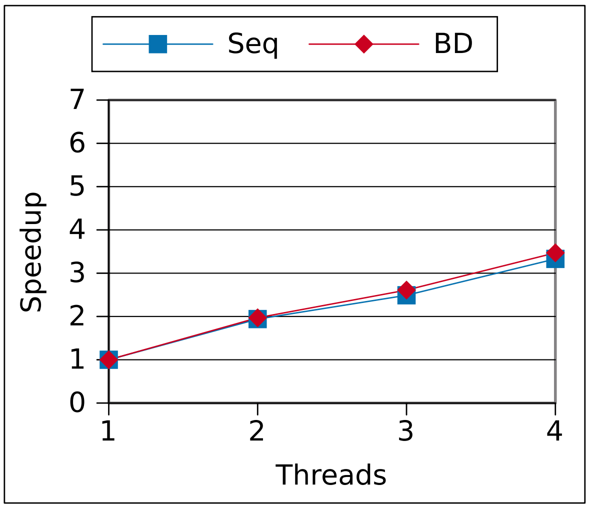
Discrete discretizes the source frame by reducing the number of color representations.

Binary creates a binary (two-colored) frame from the source. Every source pixel that is different from or above zero is set, and every source pixel that equals zero or less is unset.

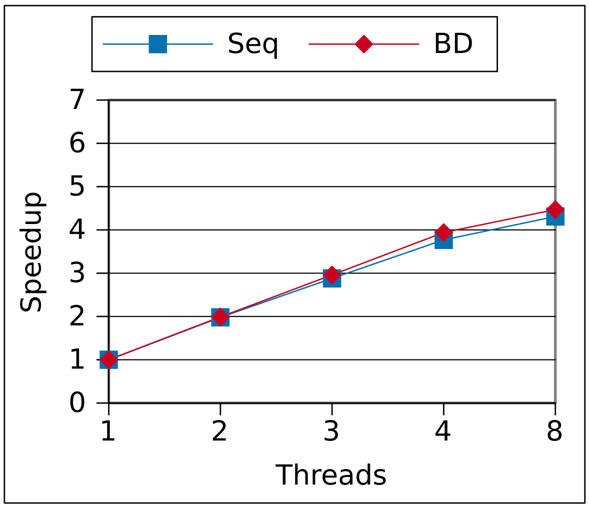
Table 2.2: Image processing filters used in experiements.

Pipeline	Filter	Seq	BD	BD-CACHED
A	Blur	9.00	162.00	9.03
	Sobel X	9.00	9.00	9.00
	Sobel Y	9.00	9.00	9.00
	Sobel Magnitude	2.00	2.00	2.00
	Threshold	1.00	1.00	1.00
B	Undistort	4.00	10.24	2.57
	Rotate 6°	3.78	2.56	2.56
	Crop	1.00	1.00	1.00
C	Undistort	4.00	8.15	2.04
	Rotate 60°	2.59	2.04	2.04
	Crop	1.00	1.00	1.00
D	Discrete	1.00	1.00	1.00
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00
E	Threshold	1.00	3.19	0.80
	Binary	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
F	Threshold	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
	Binary	1.00	1.00	1.00
G	Rotate 30°	3.19	3.19	3.19
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00

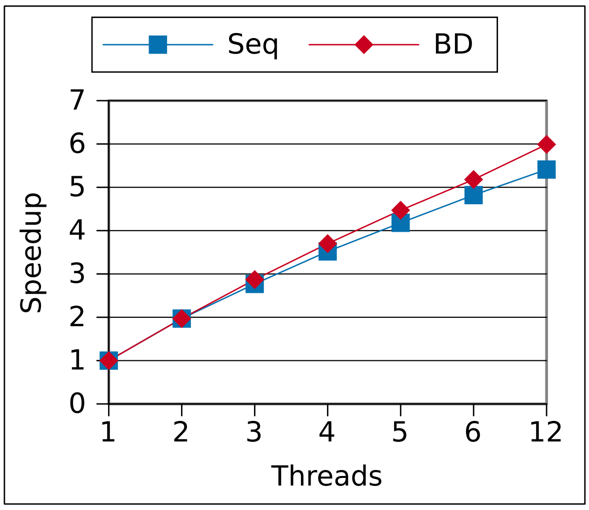
Table 2.3: Evaluated pipelines and the average number of operations performed per pixel with different execution structures. Filters are defined in table 2.2.



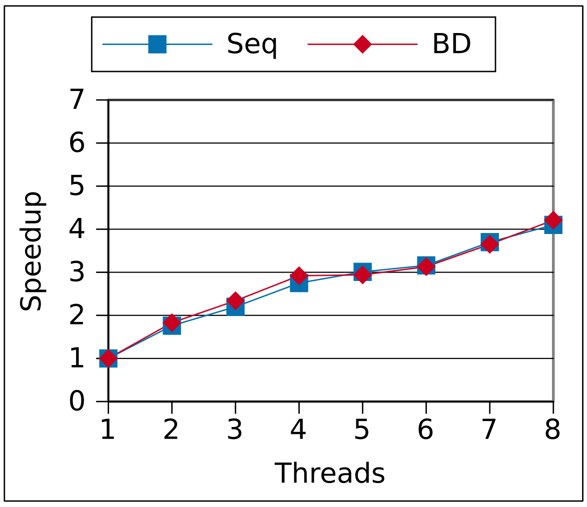
(a) Nehalem (4)



(b) Sandy Bridge (4)



(c) Sandy Bridge-E (6)

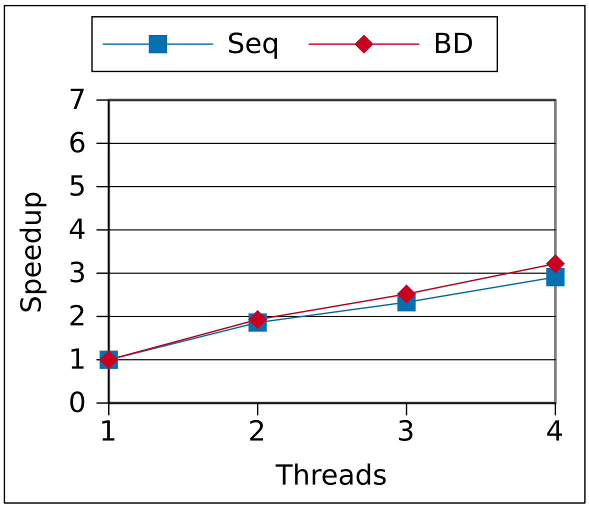


(d) Bulldozer (8)

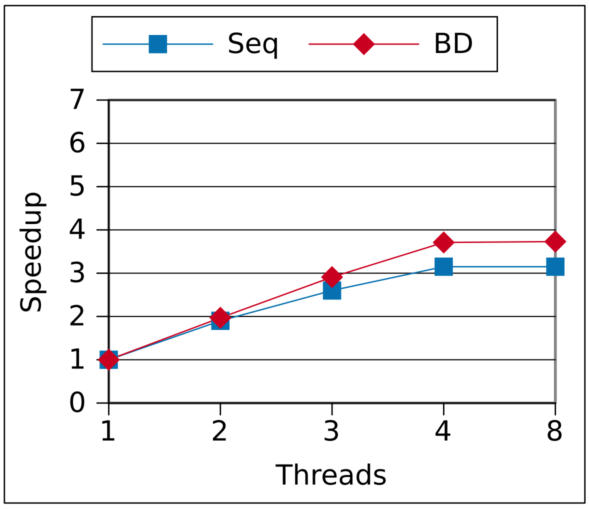
Figure 2.2: Individually normalized scalability of pipeline B running the tractor test sequence. Number of physical cores in parenthesis.

4x execution speedup. The threads created are handled by the Linux scheduler, which decides what thread to execute on which CPU (no affinity). Each new thread created works on its own frame-segment, but the last frame-segment is always processed by the thread that performs the I/O operations. In the single-threaded execution, both the I/O and processing operations are performed in the same thread, and therefore also on the same CPU core. Using two threads, we created one additional thread that is assigned the first half of a frame while the I/O thread processes the last half of the frame. We do this to minimize fetching source data from another core's private cache.

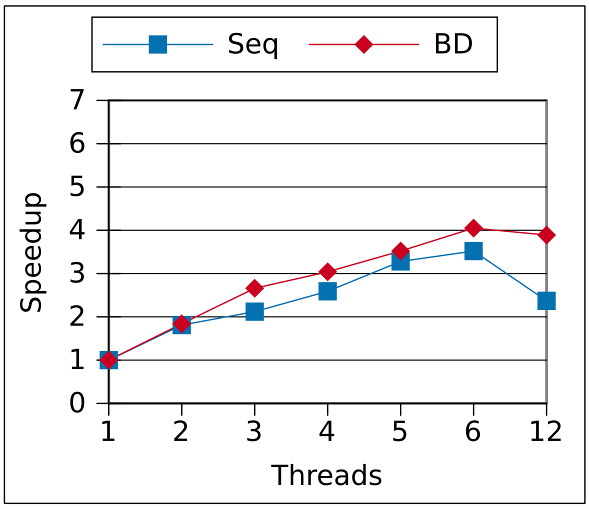
Figure 2.2 shows the relative speedup for the Nehalem, Bulldozer, Sandy Bridge and Sandy



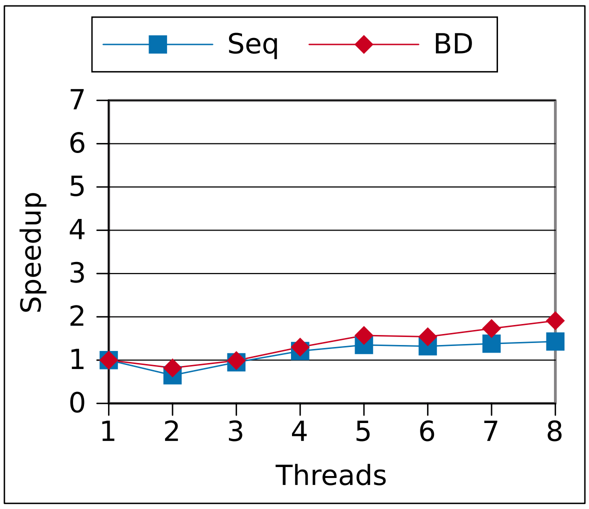
(a) Nehalem (4)



(b) Sandy Bridge (4)



(c) Sandy Bridge-E (6)



(d) Bulldozer (8)

Figure 2.3: Individually normalized scalability of pipeline B running foreman test sequence. Number of cores in parenthesis.

Bridge-Extreme microarchitectures that process pipeline B with the *tractor* sequence as input, comparing the sequential (Seq) and BD executions. Note that the running times of sequential and BD are individually normalized to their respective performance on one core, i.e., a higher value for BD does not necessarily imply better absolute performance than sequential. Looking at each architecture individually, we see that there is little difference in how the two execution methods scale on physical cores, but comparing architectures, we see that SB (figure 2.2(b)) is the only architecture that is able to scale pipeline B perfectly with the number of physical cores, as one could expect. SB-E (figure 2.2(c)) performs a little below perfect linear scaling achieved by its predecessor for this workload, and we see slightly better scalability results for BD execution than for sequential execution. On Nehalem (figure 2.2(a)), this pipeline doubles its performance with two threads, but after this, the increase in speedup per core diminishes. Bulldozer results (figure 2.2(d)) are even worse; from one to four threads we gain a 3x speedup, but there is not much to gain from using five to eight threads as we only manage to achieve a 4x speedup using the maximum number of cores. Bulldozer has four modules, each with two cores, but each module has only one FPU, and it is likely that this clamps performance to 4x. To summarize, the scalability results of pipeline B with the tractor sequence as input scales relatively well on Nehalem, Sandy Bridge and Sandy Bridge-Extreme using both execution modes. However, Bulldozer scales poorly as it only manages a 4x speedup using all of its eight cores.

Looking at scalability when testing pipeline B with the *foreman* sequence, the results become far more interesting as seen in figure 2.3. None of the four microarchitectures are able to achieve perfect scalability, and standing out is the Bulldozer plot in figure 2.3(d), where we see very little or no speedup at all using more threads. In fact, the performance worsens going from one to two threads. We look more closely into the behaviour of Bulldozer later. Furthermore, we can see that the BD mode scales better than sequential for the other architectures. From one to two threads, we get close to twice the performance, but with more threads, the difference between sequential and BD increases.

To explain why pipeline B scales much worse with foreman as input than tractor, one must take the differences into account. The foreman sequence's resolution is 352x288 (YUV, 4:2:0), which leads to frame sizes of 594 kB stored as floats. A large chunk of this can fit in private cache on the Intel architectures, and a full frame on Bulldozer. A frame in the tractor sequence has a resolution of 1920x1080 and requires almost 12 MB of data, exceeding even L3 size. In all pipeline stages except the last, data produced in one stage are referenced in the next, and if the source data does not reside in a core's cache, it leads to high inter-core traffic. Segmenting the frames per thread, as done when executing the pipelines in parallel, reduces the segments' sizes enough to fit into each core's private cache for the foreman frames, but not the larger tractor frames. Not being able to keep a large part of the frame in a core's private cache require continuous fetching from shared cache and/or memory. Although this takes time, it is the normal mode of operation. When an large part of a frame such as foreman fits in private cache after I/O, other cores that shall process this will have to either directly access the other core's private cache or request eviction to last-level cache on the other core, both resulting in high inter-core traffic. We were unable to find detailed information on this behaviour for the microarchitectures, but we can observe that scalability of this behaviour is much worse than that of the HD frame experiment. Moreover, since the BD mode only creates one set of worker threads that are used throughout the lifetime of that pipeline cycle, and it does its computations in a recursive manner, the input data is likely to reside in the core's private cache. Also, since the BD mode does not require storing intermediate results and as such does not pollute the caches, we can see it scales better than sequential for the foreman tests.

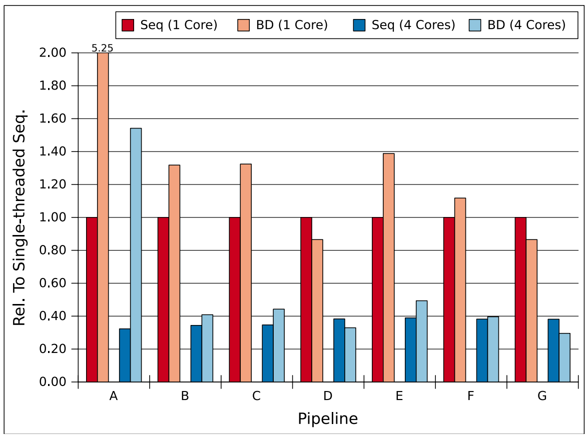
With respect to the other pipelines (table 2.3), we observed similar scalability across all architectures for both execution modes and inputs as seen in figure 2.2 and 2.3. In summary, we have shown that the BD mode provides slightly better scaling than sequential execution for our data-intensive pipelines on all architectures when the working unit to be processed (in this case a segment of a video frame) is small enough to a large extent reside in a core's private cache. Although scalability is beneficial, in the next section, we will look at the performance relative to sequential execution and see how BD and sequential perform against each other.

Performance

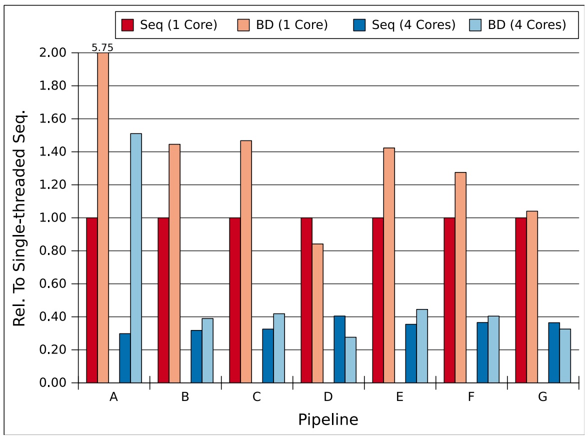
Our experiments have shown that achieving linear scaling on our data-bound filters is not trivial, and we have seen that it is especially hard for smaller units of work that mostly fit into a core's private cache and needs to be accessed on multiple cores. In addition, there are large microarchitectual differences visible. In this section, we look at the various pipelines as defined in table 2.3 to see how the sequential execution structure compares to backward dependency on various microarchitectures.

To compare sequential and BD execution, we have plotted computation time for pipeline A to G relative to their individual single-threaded sequential execution time using the foreman test sequence in figure 2.4. The plot shows that sequential execution provides best performance in most cases, but with some notable exceptions. The BD execution structure does not store and reuse intermediate pixels in any stage of the pipeline. Thus, when a pixel is accessed multiple times, it must be regenerated from the source. For example, if a filter in the last stage of a pipeline needs to generate the values of a pixel twice, every computation in every stage in the current pipeline involved in creating this output pixel must be executed twice. Obviously, this leads to a lot of extra computation as can be seen from table 2.3, where for instance the source pixels are accessed 162 times per pixel for the BD approach in pipeline A, but only 9 times for the sequential approach. This is reflected in the much higher BD computation time than sequential for pipeline A for all plots in figure 2.4.

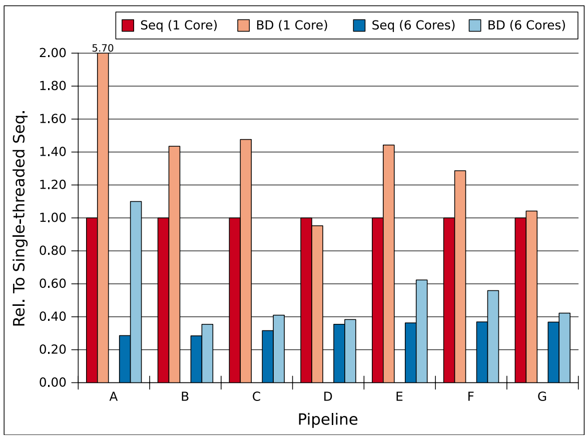
When looking at the other pipelines, we can see significant architectural differences. Again,



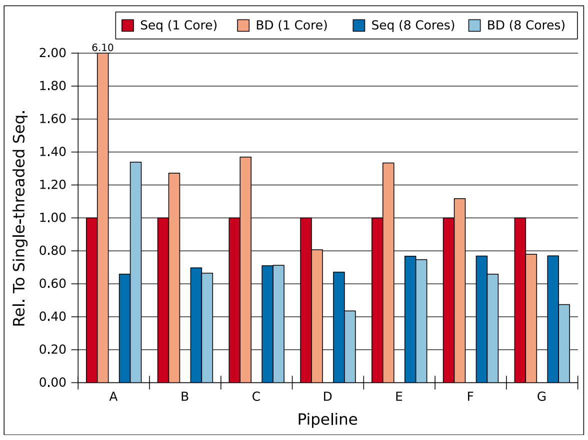
(a) Nehalem



(b) Sandy Bridge



(c) Sandy Bridge-E



(d) Bulldozer

Figure 2.4: Execution structure performance using the foreman dataset. Running times are relative to each pipeline's 1-core sequential time. Lower is better.

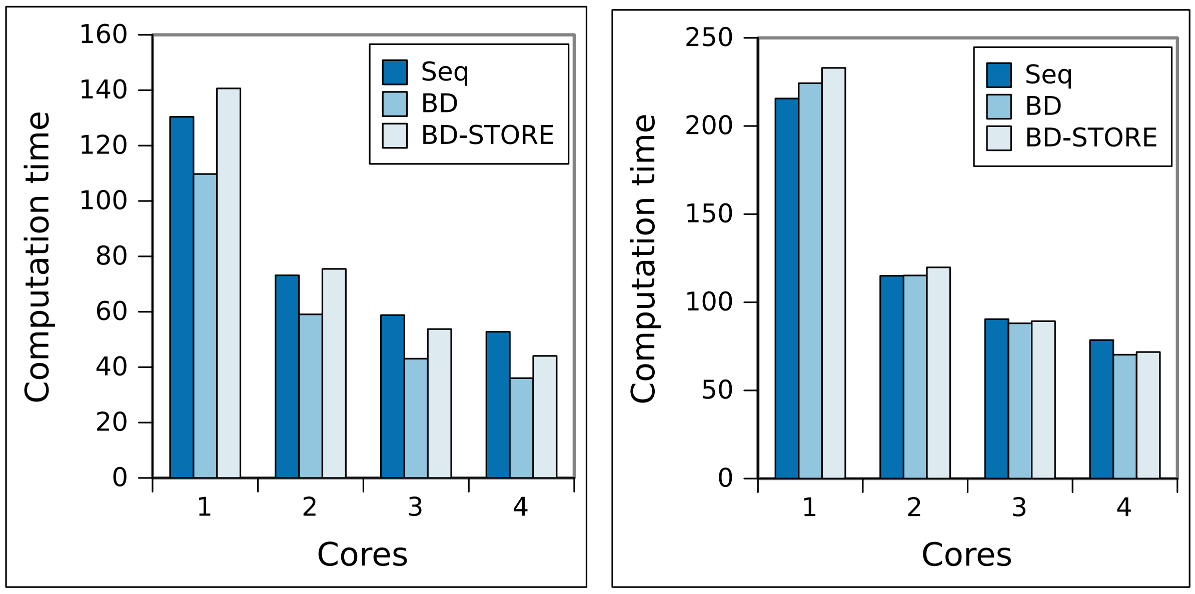
Bulldozer stands out showing that for pipeline D, F, and G, the BD approach performs considerably better than sequential execution using all eight cores. Pipeline D and G perform better with BD execution, but it is rather unexpected for pipeline F, which does require re-computation of many intermediate source pixels. Still, the scalability achieved on Bulldozer for the foreman sequence was small, as we saw in the scalability section.

The next observation we see is the performance of pipeline D. This pipeline performs the same amount of work regardless of execution structure, since every intermediate pixel is accessed only once. Most architectures show better performance for the BD approach, both for one and all cores. The only exception is the Sandy Bridge-E, which performs slightly worse than sequential when using 6 cores. A similar behaviour as pipeline D is to be expected from pipeline G since it requires the same amount of work for both modes. This turns out not to be the case; for single-threaded execution on Nehalem and Bulldozer, pipeline G is faster using BD. Using all cores, also Sandy Bridge performs better using the BD approach. Sandy Bridge-E runs somewhat faster with sequential execution. We note that Sandy Bridge and Sandy Bridge-E are very similar architectures, which ought to behave the same way. This turns out not to be the case, and even this small iteration of the microarchitecture may require different low-level schedules to get the highest level of performance - even for an embarrassingly parallel workload.

To find out if the performance gain seen with the BD approach is caused by better cache usage by keeping intermediate data in private caches or by the reduction of cache and memory pollution resulting from not storing intermediate results, we looked at pipeline D and G using BD while storing intermediate data. This mimics the sequential approach, although data is not referenced again later on, resulting in less cache pressure. Looking at figure 2.5, which shows pipeline D and G for Sandy Bridge, we can see that for a single core, the overhead of writing back intermediate results using BD-STORE results in worse performance than sequential execution, whereas this overhead diminishes when the number of threads is increased. Here, the BD-STORE structure outperforms sequential execution significantly. Accordingly, we ascribe the performance gain for the BD approach in this case to better private cache locality and usage than sequential execution.

Backward Dependency with Buffer Cache

Having shown that backward dependency execution structure performs better than sequential execution in some cases, we look at ways to improve the performance further. The main drawback of BD execution is, as we saw, that intermediate pixels must be recomputed when accessed



(a) Pipeline D (b) Pipeline G

Figure 2.5: Computation time (ms) for foreman on Sandy Bridge storing (BD-STORE) intermediate results, but not referencing stored results.

multiple times, evident by pipeline A results. We have also shown that, when doing the same amount of work as the sequential approach, BD can perform better such as with pipeline D and G. In this section, we experiment with adding a buffer cache that keeps intermediate data for later reuse to mitigate this issue.

Instead of recursively generating all prior intermediate data when a pixel is accessed, the system checks a data structure to see if it has been accessed earlier. To do this without locking, we set a bit in a bitfield atomically to mark a valid entry in the buffer cache. It is worth noting that this bitfield consume a considerably amount cache space by using one bit for every pixel, e.g., about 37 kB for every stage in a pipeline for the foreman sequence. Other approaches for this buffer cache are possible, including a set of ranges and tree structures, but this is left for further work.

The cached results for pipelines A to G (BD-CACHED) using four threads on Sandy Bridge and processing foreman are shown in figure 2.6. We can see that the BD-CACHED approach provides better performance than sequential on pipeline B, D and G, but only B performs better than BD. Pipeline B has rotation and crop stages that reduce the amount of intermediate pixels that must be produced from the early filters compared to sequential execution (see table 2.3). In comparison, pipeline C has also a significant reduction of intermediate pixel requirements, but we do not see a similar reduction in computation time. The only difference between pipeline B

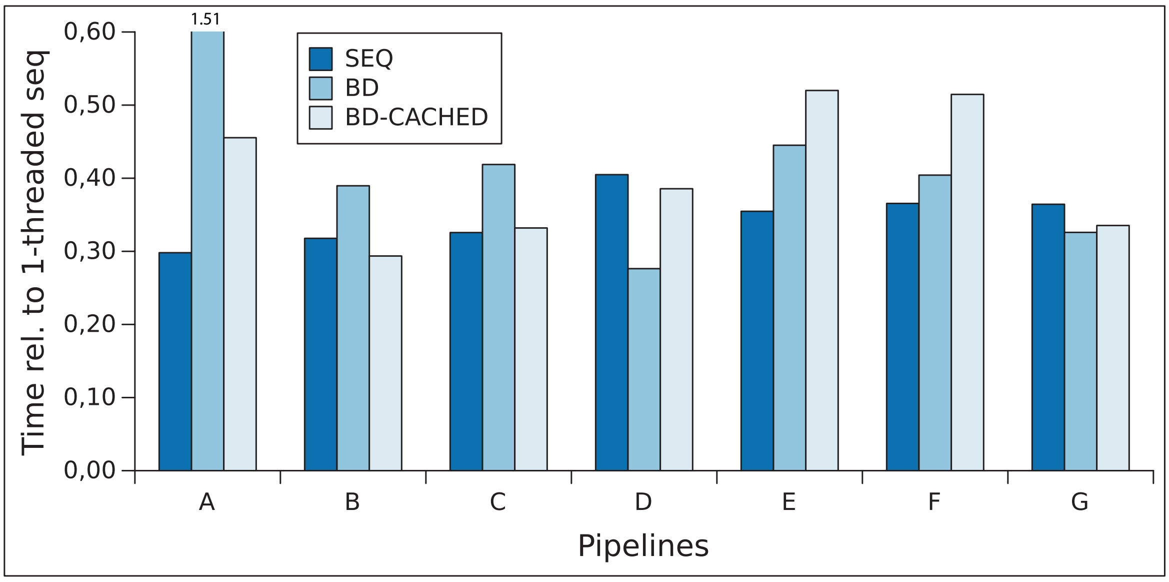


Figure 2.6: Computation time (ms) for Backward Dependency with buffer cache measured relative to single threaded sequential and tested with Sandy Bridge using 4 threads and foreman as input sequence.

and C is the amount of rotation, with 6° for B and 60° for C. The skew from rotation for the pipelines causes long strides over cache lines when accessing neighbouring pixels from previous stages, which can be hard to prefetch efficiently. Also, parts of an image segment processed on a core may cross boundaries between segments, such that high inter-core traffic is required with sequential execution even though the same core processes the same segment for each stage. This is avoided with BD and BD-CACHED modes, but we can not predict what mode performs better in advance.

Affinity

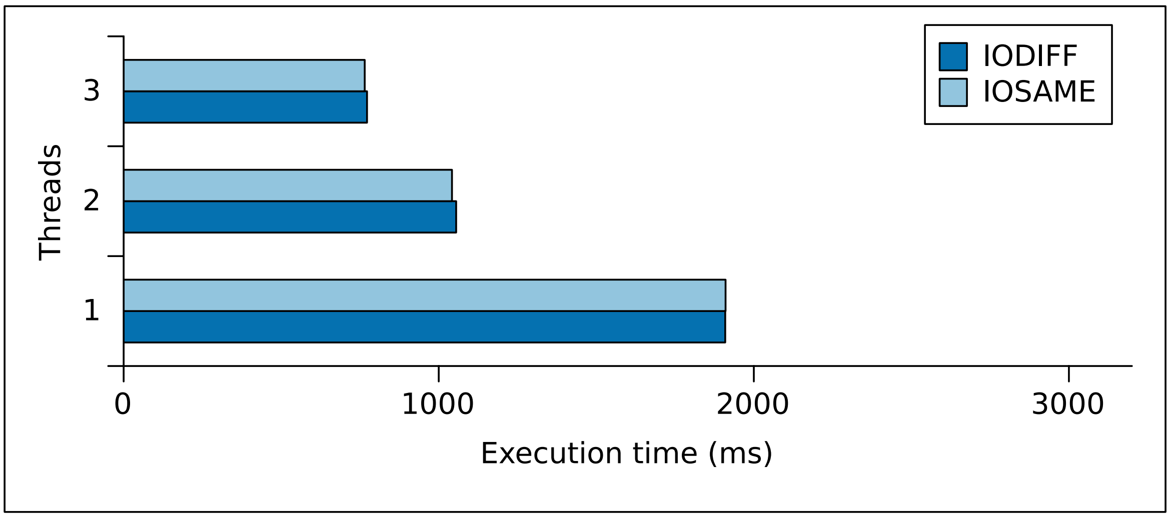
In the scalability section, we saw the diminishing performance when processing the foreman sequence with two and three threads on the Bulldozer architecture. When processing our pipelines in a single thread, we did not create a separate thread for processing, but processed in the thread that handled the I/O operations. Further, this lack of scaling was only observed when processing the low-resolution foreman sequence, for a large part of the frame could fit inside a core's private L2 cache. To investigate this further, we did an experiment where we manually specified each thread's affinity, which implies that we use separate threads doing I/O and processing, even in single-threaded execution.

Even though we only noticed the lack of scaling on the Bulldozer architecture, we tested the impact of affinity on the Nehalem and Sandy Bridge architectures as well (we omitted SB-E, which is very similar to Sandy Bridge). In figure 2.7, we have plotted the results for sequential processing of pipeline B while varying the number of threads. On Nehalem and Sandy Bridge, we tested two different execution modes, one mode where the processing of a frame was done on the same core that executed the I/O thread (IOSAME), while the second mode processed the frame on a different core than the I/O thread (IODIFF). Since the Bulldozer architecture has CPU modules, we added an additional mode for this architecture when processing threads were executed on different CPU modules than the thread handling I/O (MODULEDIFF).

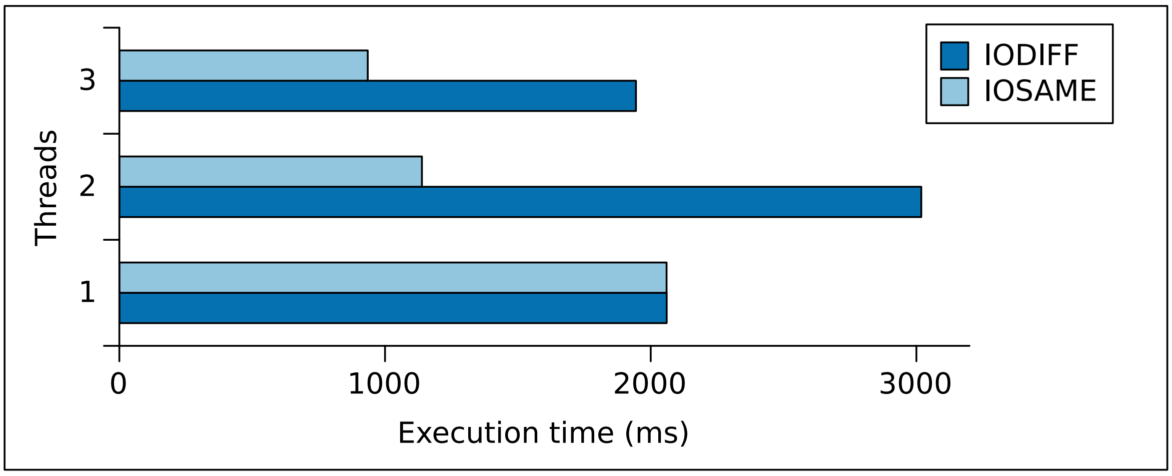
We expected that processing on another core than I/O would increase the processing time, which was found to be only partially correct: From figure 2.7(a) we see that there are not any significant difference in processing on the same core as the I/O thread versus a different core on Sandy Bridge. Much of the same behaviour is seen on Nehalem using one thread, but when using two threads there is a large penalty of pinning these to different cores than the one doing I/O operations. In this case, using two threads and executing them on different cores than the I/O operations add so much to the computation time that it is slower than the single threaded execution.

Looking at the Bulldozer architecture in figure 2.7(c), using only one thread we see that there are not any significant difference on what core or CPU module the processing thread is placed on. However as with Nehalem, when using two or more threads the IODIFF and MODULEDIFF execution modes have a huge negative impact on the performance. Even more unexpected, IOSAME and IODIFF using three threads are actually slower than IOSAME using two threads, and the fastest execution using eight threads completes in 1174 ms (omitted in plot), not much faster than what we can achieve with two threads.

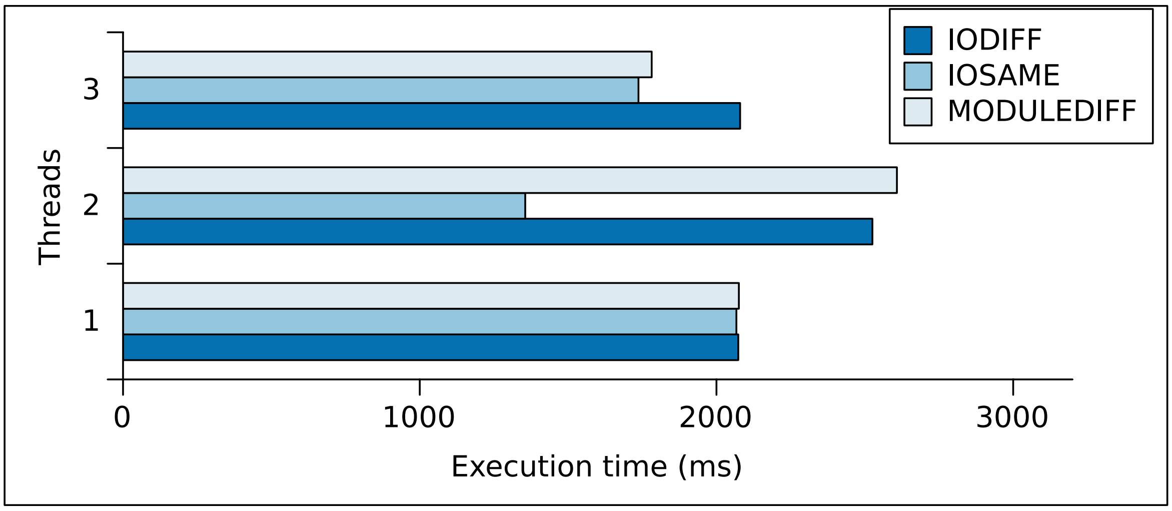
In summary, we have seen that thread affinity has a large impact on Nehalem and Bulldozer when a large part of the data fits into the private cache. This can cause the two threads to have worse performance than a single thread when the data reside in another core's private cache. On Sandy Bridge, this limitation has been lifted and we do not see any difference due to affinity. Bulldozer is unable to scale to much more than two threads when the dataset fits into the private cache, presumably because the private cache is shared between two cores and the cost of doing inter-module cache access is too high.



(a) Sandy Bridge



(b) Nehalem



(c) Bulldozer

Figure 2.7: Execution times (ms) of pipeline B for the foreman video using different thread affinity strategies (lower is better).

Discussion

The long-running data-intensive filters described in this section deviate from typical workloads when looking at performance in terms of the relatively little computation required per unit of data. The filters used are primitive, but we are able to show large performance impacts by varying the order of execution and determining what core should do what. For such simple and embarrassingly parallel workloads, it is interesting to see the significant differences on how these filters perform on modern microarchitectures. As a case study, we chose image processing algorithms that can intuitively be connected in pipelines. Real-world examples of such pipelines can be found in OpenCV [8], node-based software such as the Nuke [24] compositing tool and various VJ software (digital effects at live events). Other signal processing domains such as sound processing are applicable as well.

There is a big difference between batch processing and processing a stream of data. In the former, we can spread out independent jobs to a large number of cores, while in the latter we can only work on a limited set of data at a time. If we batch-processed the pipelines, we could instantiate multiple pipelines, each processing frames independently while avoiding the scalability issues that we experienced with foreman. In a streaming scenario, however, this is not possible.

The results shown in this work are unexpected and confusing. We had not anticipated such huge differences in how these modern processors perform with our workloads. With the standard parallel approach for such applications with either sequential execution of the pipeline or a work stealing approach, it is apparent that there is much performance to be gained by optimizing the order of operations for better cache usage. After all, we observe that the performance of the execution modes varies a lot with the behavior of a filter, e.g, amount of rotation applied. Moreover, the computation time varies inconsistently with the number of threads, e.g., performance halved with two threads and sub-optimal processor affinity compared to a single thread. Thus, scheduling these optimally based on a-priori knowledge, we conjecture, is next to impossible, and profiling and feedback to the scheduler must be used to find the best configuration.

One option is to use profile-driven optimizations at compile time [25], where one or more configurations are evaluated and later used during runtime. This approach does, however, not work with dynamic content or tasks, i.e., if the parameters such as the rotation in pipeline B changes, the system must adapt to a new configuration. Further, the preferred configuration may even change based on interactions between co-scheduled tasks, e.g., a CPU-bound and memory-bound task may perform better when co-scheduled than two memory bound tasks [26].

The ideal option then is to have a low-level scheduler that adapts dynamically to varying tasks, data, architectural limitations and shared resources. We propose an approach that uses instrumentation and allows a scheduler to gauge computation times of specific filters at execution time. Since the workloads we look at are periodic and long-running, the low-level scheduler can react to the observations and make appropriate adjustments, e.g., try different execution modes, data granularity, affinity and co-running competing workloads.

The prevalent approaches used in execution systems today are work stealing variants. Here, the system typically use a heuristics to increase cache locality such as spawning new tasks in the same queue; or stealing tasks from the back of another queue instead of the front to reduce cache contention, assuming tasks that are near in the queue are also near in terms of data. Although work stealing in its simplicity provides great flexibility, we have shown that the execution order has a large performance impact on filter operations. For workloads that are long-running and periodic in nature, we can expect that an adaptive low-level scheduler will outperform the simple heuristics of a work stealing scheduler. An adaptive work stealing approach is feasible, where work units are enqueued to the worker threads based on the preferred execution mode, data granularity and affinity, while still retaining the flexibility of the work stealing approach. Such a low-level scheduler is considered for further work. Another direction that we want to pursue is building synthetic benchmarks and looking at performance counters to pinpoint the cause of some of the effects that we are seeing, in particular with the Bulldozer microarchitecture.

Summary

In this case study, we have looked at run-time considerations for executing (cyclic) streaming pipelines consisting of a data-intensive filters for media processing. A number of different filters and pipelines have been evaluated on a set of modern microarchitectures, and we have found several unexpected performance implications, within the well-known x86-family of microprocessors, like increased performance by backtracking pixel dependencies to increase cache locality for data sets that can fit in private cache; huge differences in performance when I/O is performed on one core and accessed on others; and different execution modes perform better depending on minor parameter variations in filters (or stages in the processing pipeline) such as the number of degrees to rotate an image. The implication of the very different behaviors observed on the different microarchitectures is a demand for scheduling that can adapt to varying conditions using instrumentation data collected at runtime.

2.1.3 Implications

Symmetric cores allows for easy portability of applications between compatible microarchitectures. From a programmer's perspective, it is easy to see why some prefer using SMP for multimedia processing, even though the performance might not keep up with that of heterogeneous architectures. However, as we have showed in section 2.1.2, achieving high performance might still require microarchitecture considerations, even though the program is portable and runs on compatible processors. For multimedia processing, using traditional symmetric cores in combination with various accelerators is common, further complicating efficient use of the resources. As we will see in section 2.2.2, predicting what mode and parameters to use in a portable manner *a priori* is difficult. Minor variations in an implementation have a large impact on performance. Since performance is important, and especially for multimedia operations with realtime requirements, optimizing for this problem is an important goal. Since the best configuration is not known and depends on a large array of parameters, we think probing configurations combined with data collection (instrumentation) in a feedback loop is a good option, and is something we will look further into in the next section.

2.2 Asymmetric exclusive memory architectures

2.2.1 Overview

Another interesting architecture type has exclusive memory tightly coupled to some of the cores on a processor. This memory typically runs at core frequency with high bandwidth and low latency to the core or set of cores connected to it. The downside is the relatively small amount of exclusive memory each core usually has, requiring careful partitioning and planning of what data should reside where and when. Typically for this architecture is a host or main processor with different capabilities than the other cores to manage the full processor's workloads and data. Exclusive memory in these architectures can be seen as a software managed cache, where the developer or the runtime is responsible for prefetching data for processing instead of a fixed prefetching algorithm implemented in hardware. In contrast to caches, this allows flexible data management at the cost of higher programming complexity, enabling developers to only prefetch data that will actually be used if this is known in advance.

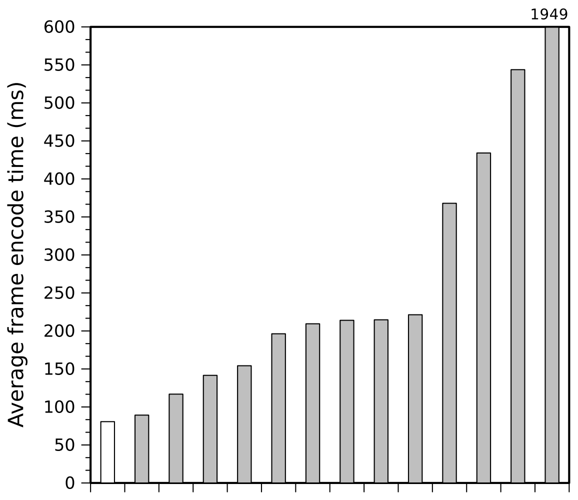
Several such architectures exist today, with prominent examples including Cell Broadband Engine [27], various graphics processors and accelerator cards. As graphics processors advanced, they have evolved from static graphics functions to highly flexible and programmable processors suitable for general purpose computing. The dominant API today is CUDA [5] by Nvidia, which exposes the GPU as a accelerator card for high performance computing. GPUs typically have several types of memory and require explicit requests to move data between them. Similarly to the Cell, this increases the responsibility of system programmers to correctly utilize memory for maximum performance. In section 2.2.2, we compare using Cell and GPUs to encode motion JPEG video as an example.

2.2.2 Case study: MJPEG on Cell and GPU

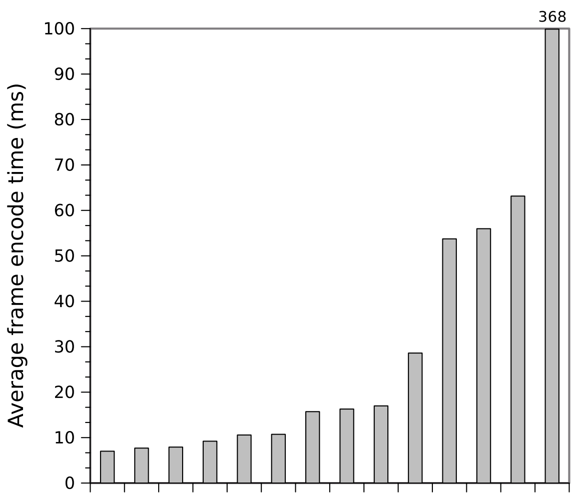
Heterogeneous systems like the STI Cell Broadband Engine (Cell) and PCs with Nvidia graphical processing units (GPUs) have recently received a lot of attention. They provide more computing power than traditional single-core systems, but it is a challenge to use the available resources efficiently. Processing cores have different strengths and weaknesses than desktop processors, the use of several different types and sizes of memory is exposed to the developer, and limited architectual resources require considerations concerning data and code granularity.

In this section, we want to learn how to *think* when the system at our disposal is a Cell or a GPU for processing multimedia workloads. We aim to understand how to use the resources efficiently, and point out tips, tricks and troubles, as a small step towards a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we have looked at effective programming for the workload-intensive yet relatively straightforward Motion-JPEG (MJPEG) video encoding task. The MJPEG encoder has many parallel opportunities, including task, data and pipeline parallelism with limited dependencies, and as such is perfect for execution on heterogeneous architectures. In the encoder a lot of CPU cycles are consumed in the sequential discrete cosine transformation (DCT), quantization and compression stages. On single core systems, it is almost impossible to process a 1080p high definition video in real-time, so it is reasonable to apply multicore computing in this scenario.

This work was first published in an article presented at NOSSDAV 2010 [28], and was written in the context of the latest graphics processors available at the time of writing. The world of heterogeneous computing is evolving rapidly with NVIDIA and AMD continuously pushing the envelope, and in terms of capabilities of the cards, much has changed since then. However, many of the key insights drawn from this work is still applicable, with the exception of hardware specific capabilities. Newer cards have improved the architecture in several ways, but the general structure and ideas remain the same.



(a) Cell on PS3 (6 SPEs)



(b) GPU on GTX 280

Figure 2.8: Frame encode time for MJPEG implementations sorted by running time.

As a baseline for how important the architectual considerations are for writing efficient programs for these types of architectures, we compared 14 different student's adaptations of the same motion jpeg video encoder; the performance numbers of these alone give a good indication of the complexity of such a task, and that achieving high performance is far from trivial. Derived from a sequential codebase, these multicore implementations differ in terms of algorithms used, resource utilization and coding efficiency.

Figure 2.8 shows performance results for encoding the ``tractor'' video clip[1] in 4:2:0 HD. The differences between the fastest and slowest solution are 1869 ms and 362 ms per frame on Cell and GPU, respectively, and it is worth noting that the fastest solutions were disk I/O-bound. To gain experience of what works and what does not, we have examined these solutions. We have not considered coding style, but revisited algorithmic choices, inter-core data communication (memory transfers) and use of architecture-specific capabilities.

In general, we found that these architectures have large potentials, but also many possible pitfalls, both when choosing specific algorithms and for implementation-specific decisions. The way of thinking cross-platform is substantially different, making it an art to use them efficiently.

Background

SIMD and SIMT Multimedia applications frequently perform identical operations on large data sets. This has been exploited by bringing the concept of SIMD (single instruction, multiple

[1]Available at ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/test_sequences/1080p/tractor.yuv

data) to desktop CPUs, as well as the Cell, where a SIMD instruction operates on a short vector of data, e.g., 128-bits for the Cell SPE. Although SIMD instructions have become mainstream with the earliest Pentium processors and the adoption of PowerPC for MacOS, it has remained an art to use them. On the Cell, SIMD instructions are used explicitly through the vector extensions to C/C++, which allow basic arithmetic operations on vector data types of intrinsic values. It means that the programmer can apply a sequential programming model, but needs to adapt memory layout and algorithms to the use of SIMD vectors and operations.

Nvidia uses an abstraction called SIMT (single-instruction, multiple thread). SIMT enables code that uses only well- known intrinsic types but that can be massively threaded. The run-time system of the GPU schedules these threads in groups (called warps) whose optimal size is hardware-specific. The control flow of such threads can diverge like in an arbitrary program, but this will essentially serialize all threads of the block. If it does not diverge and all threads in a group execute the same operation or no operation at all in a step, then this operation is performed as a vector operation containing the data of all threads in the block.

The functionality that is provided by SIMD and SIMT is very similar. In SIMD programming, vectors are used explicitly by the programmer, who may think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types.

STI Cell Broadband Engine The Cell Broadband Engine is developed by Sony Computer Entertainment, Toshiba and IBM. As shown in Figure 2.9, the central components are a Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs) connected by the Element Interconnect Bus (EIB). The PPE contains a general purpose 64-bit PowerPC RISC core,

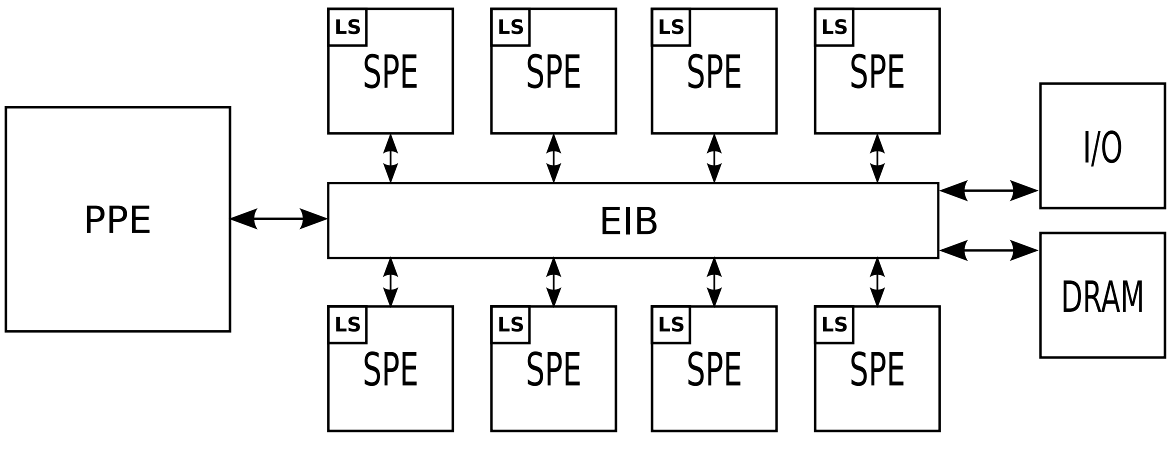


Figure 2.9: Cell Broadband Engine Architecture

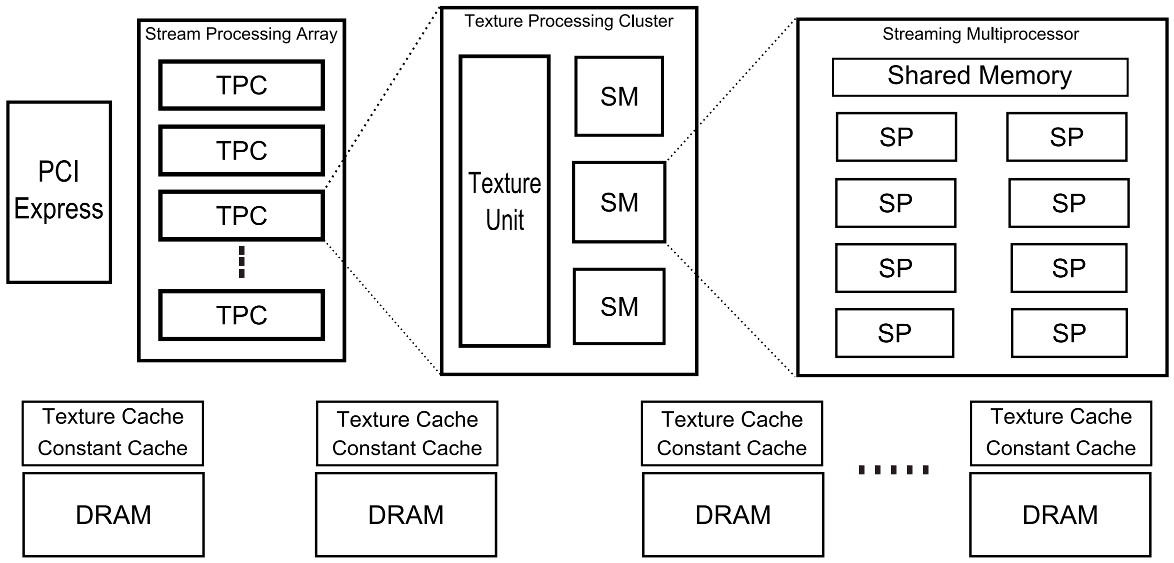


Figure 2.10: Nvidia GT200 Architecture

capable of executing two simultaneous hardware threads. The main purpose of the PPE is to control the SPEs, run an operating system and manage system resources. It also includes a standard Altivec-compatible SIMD unit. An SPE contains a Synergistic Processing Unit and a Memory Flow controller. It works on a small (256KB) very fast memory, known as the local storage, which is used both for code and data without any segmentation. The Memory Flow Controller is used to transfer data between the system memory and local storage using explicit DMA transfers, which can be issued both from the SPE and PPE.

Nvidia Graphics Processing Units A GPU is a dedicated graphics rendering device, and modern GPUs have a parallel structure, making them effective for doing general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia has released the CUDA framework with a programming language similar to ANSI C. In CUDA, the SIMT abstraction is used for handling thousands of threads.

The GPU generation used in this work is the Nvidia GT200, and is shown in Figure 2.10. The GT200 chip is presented to the programmer as a highly parallel, multi-threaded, multi-core processor - connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters (TPC) with 3 streaming multiprocessors (SM). A single SM contains 8 stream processors (SP), which are the basic ALUs for doing calculations. GPUs have other memory hierarchies than an x86 processor. Several types of memory with different properties are available. An application (*kernel*) has exclusive control over the memory. Each thread has a private local memory, and the threads running on the same stream multiprocessor (SM) have access to a shared memory. Two additional read-only memory spaces called constant and texture are available to all threads. Finally, there is the global memory that can be accessed by all threads. Global memory is not cached, and it is important that the programmer ensures that running threads perform *coalesced* memory accesses. Such a coalesced memory access requires that the threads' accesses occur in a regular pattern and creates one large access from several small ones. Memory accesses that cannot be combined are called uncoalesced.

Experiments

By learning from the design choices of the implementations in Figure 2.8, we designed experiments to investigate how performance improvements were achieved on both Cell and GPU. We wanted to quantify the impact of design decisions on these architectures.

All experiments encode HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* test sequence. However, we used only the first frame of the sequence and encode it 1000 times in each experiment to overcome the disk I/O bottleneck limit. This becomes apparent at the highest level of encoding performance since we did not have a high bandwidth video source available. All programs have been compiled with the highest level of compiler optimizations using gcc and nvcc, respectively, for Cell and GPU. The Cell experiments have been tested on a QS22 bladeserver (8 SPEs, the results from Figure 2.8 were on a PS3 with 6 SPEs) and the GPU experiments on a GeForce GTX 280 card.

Motion JPEG Encoding The MJPEG format is widely used by webcams and other embedded systems. It is similar to videocodecs such as Apple ProRes and VC-3, used for video editing and post-processing due to their flexibility and speed, hence the lack of inter-prediction between frames. As shown in Figure 2.11, the encoding process of MJPEG comprises splitting the video frames in 8x8 macroblocks, and each of them must be individually transformed to the frequency domain by forward discrete cosine transform (DCT) and quantized before the output is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC. Our encoder uses predefined Huffman tables for compression of the DCT coefficients of each macroblock. The VLC step is not context adaptive, and macroblocks can thus be compressed independently. The length of the resulting bitstream, however, is probably not a multiple of eight, and most such blocks must be bit-shifted completely when the final bitstream is created.

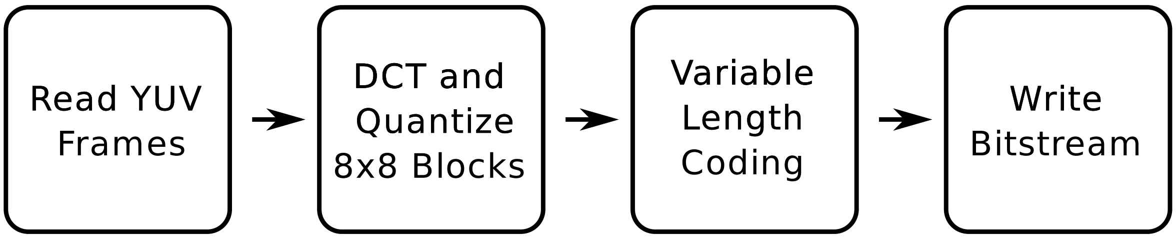


Figure 2.11: Overview of the MJPEG encoding process

The MJPEG format provides many layers of parallelism; starting with the many independent operations of calulating DCT, the macroblocks can be transformed and quantized in arbitrary order, also frames and color components can be encoded separately. In addition, every frame is entropy-coded separately. Thus, many frames can be encoded in parallel before merging the resulting frame output bitstreams. This gives a very fine-level granularity of parallel tasks, providing great flexibility in how to implement the encoder. It is worth noting that many problems have much tighter data dependencies than we observe in the MJPEG case, but the general ideas for optimizing individual parts pointed out in this thesis stand regardless of whether the problem is limited by dependencies or not.

The forward 2D DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D-plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label 1D-plain uses two consecutive 1D transformations with a transpose operation in between and after. This avoids symmetries, and the 1D transformation can be optimized further. One optimization uses the AAN algorithm, originally proposed by Arai et al. [29] and further refined by Kovac and Ranganathan [30]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the postscale operation, and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [31].

More algorithms for calculating DCT exist, but they are not covered here. We have implemented the different DCT algorithms as scalar single-threaded versions on x86 (Intel Core i5 750). The performance details for encoding HD video were captured using oprofile and can be seen in Figure 2.12. The plot shows that the 1D-AAN algorithm using two transpose operations was the fastest in this scenario, with the 2D-matrix version as number two. The average encoding time for a single frame using 2D-plain is more than 9 times that of a frame encoded using 1D-AAN. For all algorithms, the DCT step consumed most CPU cycles.

Cell Broadband Engine Experiments Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts is available for mapping the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but also the entropy coding step can run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores, and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We solved this by adding a frame queue between the frame reader and the DCT step, and another queue between the DCT and VLC steps. Since a frame is processed in full by a single processor, the AAN algorithm is well suited for the Cell. It can be implemented in a straight-forward manner for running on SPEs, with VLC coding placed on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD) resulting in double encoding throughput, which can be seen in Figure 2.13 (Scalar- and Vector/PPE).

Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk in addition to dis-

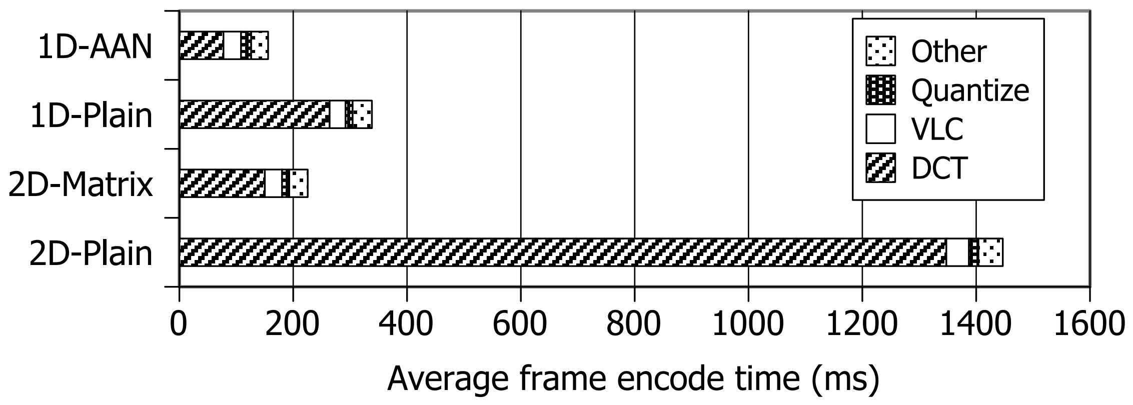


Figure 2.12: MJPEG encode time on single thread x86

patching jobs to SPEs. To be able to do this, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, i.e., two rows of luma and a single row of both chroma channels. The results show that the previous encoding speed was limited by the VLC as can be seen in Figure 2.13 (Scalar- and Vector/SPE).

To get some insight into SPE utilization, we collected a trace (using pdtr, part of IBM SDK for Cell) showing how much time is spent on the encoding parts. Figure 2.14 shows the SPE utilization when encoding HD frames for the Scalar- and Vector/SPE from Figure 2.13. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand write SIMD intrinsics to obtain high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version, and additionally 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

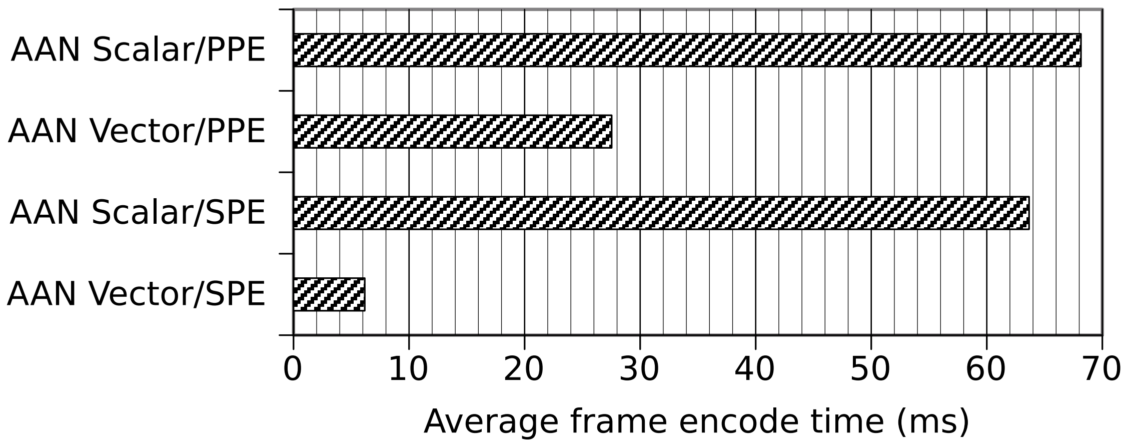


Figure 2.13: Encoding performance on Cell with different implementations of AAN and VLC placement

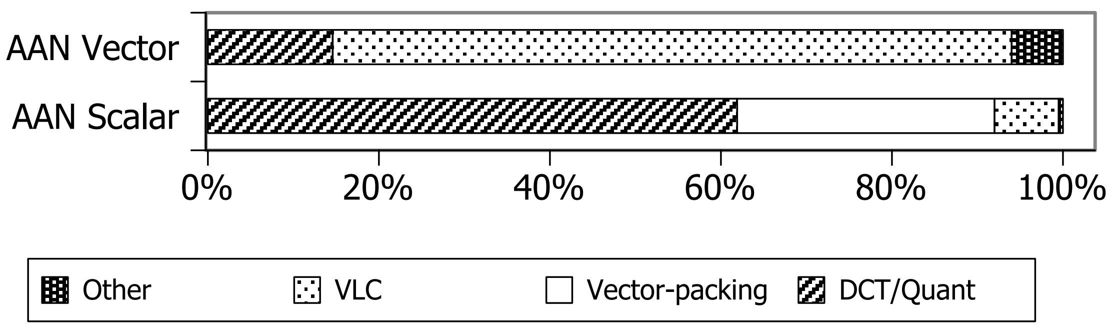


Figure 2.14: SPE utilization using scalar or vector DCT

With the vector version of DCT and quantization, the VLC coding uses about 80 % of each SPE. This can possibly be optimized further, but we did not pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to get high performance on this architecture. In particular, correctly implementing an algorithm using vector intrinsics is imperative. Of the 14 implementations for Cell in Figure 2.8, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization, and the Vector/SPE implementation in Figure 2.13 is a combination of these two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit-shifting of large fields on Cell as well as GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles, as seen in Figure 2.12, and the awkward process of gathering profiling data on multicore systems later on. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work, and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are loosely coupled. In that way, they can both be replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 2.8 to find out what differentiates the fastest from the medium speed implementations, we found some distinguishing features: The most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfers and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed, DMA operations that left SPEs in a stalled state waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture, and we have only touched upon a few of them. Some of these are obvious, but not all, and to get acceptable performance out of a program running on the Cell architecture may require multiple iterations, restructuring and even rewrites.

GPU Experiments As for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Coalesced memory accesses are known to have large performance impacts. However, few

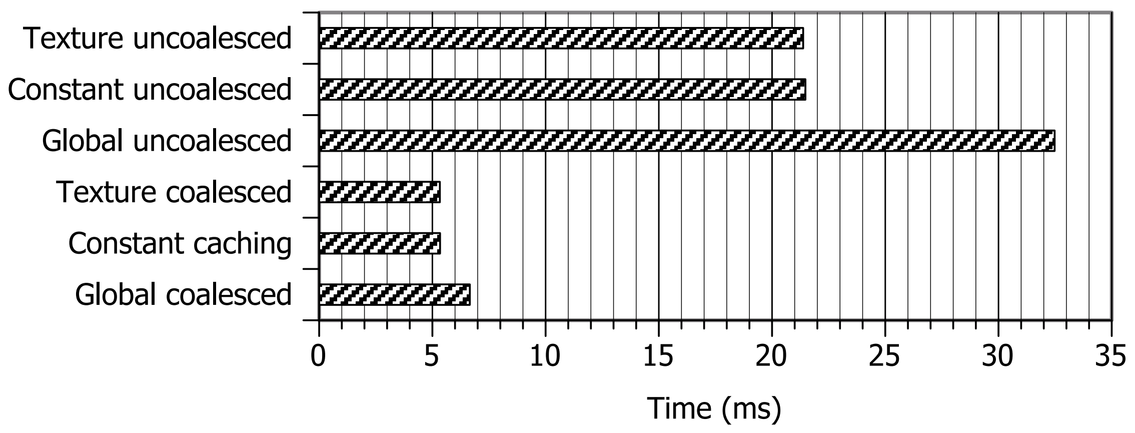


Figure 2.15: Optimization of GPU memory accesses

quantified results exist, and efficient usage of memory types, alignment and access patterns remains an art. Weimer et al. [32] experimented with bank conflicts in shared memory, but to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and write data to and from memory with both un-coalesced and coalesced access patterns [33], and used the Nvidia CUDA Visual Profiler to isolate the GPU-time for the different kernels.

Figure 2.15 shows that an uncoalesced access pattern decreases throughput in the order of four times due to the increased number of memory transactions. Constant and texture memory are cached, and the performance for uncoalesced accesses to them is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative even when cached memory types are used. It is also important to make sure the memory accesses are correct according to the specifications of particular GPUs because the optimal access patterns vary between GPU generations.

To find out how memory accesses and other optimizations affect programs like a MJPEG encoder, we experimented with different DCT implementations. Our baseline DCT algorithm is the 2D-plain algorithm. The only optimizations in this implementation are that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. As we observed in Figure 2.15, cached memory spaces improve performance compared to global memory, especially when memory accesses are uncoalesced. The second implementation, referred to as 2D-plain optimized, is tuned to run efficiently, using principles from the CUDA Best Practices Guide [34]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using boolean arithmetics and manual loop unrolling. Our third implementation, the 1D-AAN algorithm, is based upon the scalar implementation used on the Cell. Every macroblock is processed with eight threads, one thread per row of eight pixels. The input image is stored in cached texture memory and shared memory is used for temporarily storing data during processing. Finally, we look at the 2D-matrix DCT using matrix multiplications where each matrix element is computed by a thread. The input image is stored in cached texture memory, and shared memory is used for storing data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized, which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU were used. Transferring data over the PCI Express bus is expensive, and asynchronous transfers help us reuse the kernels and hide some of the PCI Express latency by transferring data in the background.

To isolate the DCT performance, we used the CUDA Visual Profiler. The profiling results of the different implementations can be seen in Figure 2.16, and we can observe that the 2D-plain optimized algorithm is faster than AAN. The 2D-plain algorithm requires significantly more computations than the others, but by correctly implementing it, we get almost as good performance as with the 2D-matrix. The AAN algorithm, which does the least amount of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

This experiment shows that for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the theoretical complexity of an algorithm.

The last GPU experiment considers entropy coding on the GPU. As for the Cell, VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host, or by using atomic OR on the GPU. For the experiments here, we chose the former since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 2.17 shows the results of an experiment

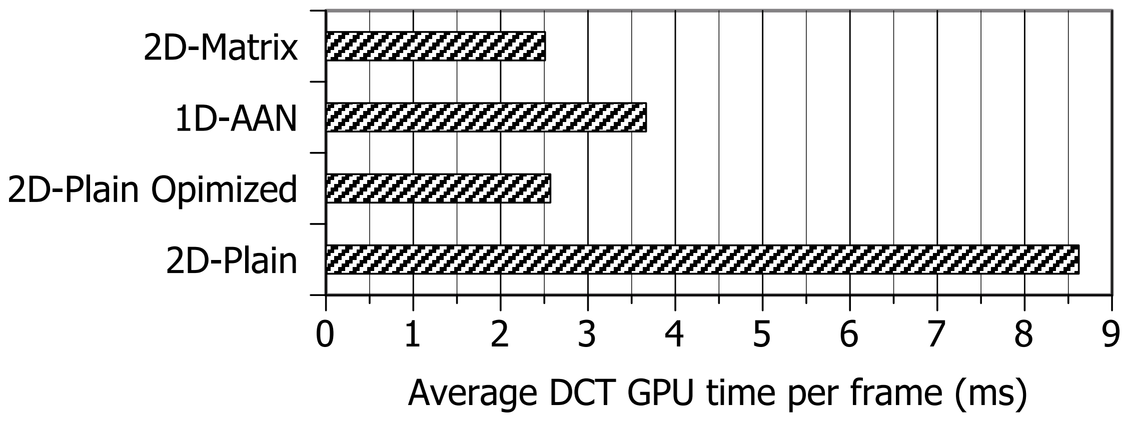


Figure 2.16: DCT performance on GPU

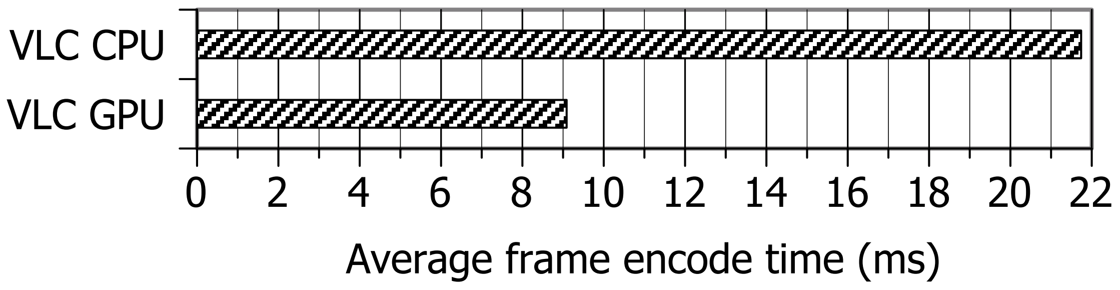


Figure 2.17: Effect of offloading VLC to the GPU

that compares MJPEG with AAN DCT with VLC performed on the host and on the GPU, respectively. We achieved a doubling of the encoding performance when running VLC on the GPU. In this particular case offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or faster GPU, the encoder will still scale.

Discussion

Heterogeneous architectures like Cell and GPU provide large amounts of processing power, and achieving encoding throughputs of 480 MB/s and 465 MB/s, respectively, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions, and that there may be large differences in the way a programmer must think.

The architectures of GPU and Cell are very different, and in this respect, some algorithms may be more suited than others. This can be seen in the experiments, where the AAN algorithm for DCT calculation performed best on both x86 and Cell, but did not achieve the highest throughput on GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

The programming models used on Cell and GPU mandate two different ways of thinking parallel. The approach of Cell is very similar to multi-threaded programming on x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches, and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads, and only synchronization through barriers. This hides the architecture complexity, and is therefore a simpler concept to grasp for some programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figure 2.8. However, to get the highest possible performance, the programmer must also understand the small details of the architecture to avoid pitfalls like warp divergence and uncoalesced memory accesses.

Deciding at what granularity the data should be partitioned is very hard to do correct *a priori*. The best granularity for a given problem differs with the architecture and even different models of the same architecture. One approach towards accomplishing this is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data decomposition, task decomposition is hard to do correctly in advance. In general, a rule of thumb is to write modular code to allow moving the parts to other cores. Also, a fine granularity is beneficial, since small modules can be merged again, and also be executed repeatedly with small overhead. Offloading is by itself advantageous as resources on the main processor become available for other tasks. It also improves scalability of the program with new generations of hardware. In our MJPEG implementations, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance on both Cell and GPU, but it may not always be the case that offloading provides higher throughput.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort for accomplishing this throughput was much higher on the Cell and was mainly caused by the tedious process of writing a SIMD version of the encoder. Porting the encoder to the GPU in a straight-forward manner without significant optimizations for the architecture yielded a very good offloading performance compared to native x86. This indicates that the GPU is easier to use, but to reap the full potential of the architecture, one must have the same deep level of understanding as with the Cell architecture.

Related work

Heterogeneous multi-core platforms like the Cell and GPUs have attracted a considerable amount of research that aims at optimizing specific applications for the different architectures such as [35] and [36]. However, little work has been done to compare general optimization details of different heterogeneous architectures. Amesfoort et al. [37] have evaluated different multicore platforms for data-intensive kernels. The platforms are evaluated in terms of application performance, programming effort and cost. Colic et al. [38] look at the application of optimizing motion estimation on GPUs and quantify impact of design choices. The workload investigated in this work is different from the workload we benchmarked in our experiments, but they show a similar trend as our GPU experiments. They also conclude that elegant solutions are not easily achievable, and that it takes time, practice and experience to reap the full potential of the architectures. Petrini et al. [39] implement a communication-heavy radiation transport problem on Cell. They conclude that it is a good approach to think about problems in terms of five dimensions and partitioning them into: process parallelism at a very large scale, thread-level parallelism that handles inner loops, data-streaming parallelism that double-buffers data for each loop, vector parallelism that uses SIMD functions within a loop, and pipeline parallelism that overlaps data access with computations by threading. From our MJPEG implementations we observed that programmers had difficulties thinking parallel in two dimensions. This level of multi-dimensional considerations strengthens our statement that intrinsic knowledge of the system is essential to reap full performance of heterogeneous architectures.

Summary

Heterogeneous, multicore architectures like Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial, and in order to learn how to think and use the resources efficiently, we have experimentally evaluated several issues to find the tricks and troubles.

In general, there are some similarities, but the way of thinking must be substantially different - not only compared to an x86 architecture, but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account both when choosing a specific algorithm and making implementation-specific decisions. A lot of trust is put on the compilers of development frameworks and new languages like Open CL, which are supposed to be a *recompile-only* solution. However, to tune performance, the application must still be hand-optimized for different versions of the GPUs and Cells available.

2.2.3 Implications

The asymmetric exclusive memory architectures provide very high performance, but at the cost of complex and time consuming software adaptations for the target architecture. Additionally, the relatively small amount of fast exclusive memory require careful planning of memory flows, and programs with low data locality may not run well on these type of architecture. Multimedia operations, such as the MJPEG encoding in section 2.2.2 are often good candidates for high performance on these with high data locality, and with algorithms that can easily be data parallelized, e.g., with vector instructions. As with the asymmetric shared memory architectures, major challenges are portability and elasticity, but memory flow to exclusive memory types requires strict attention too. One unexpected finding in the study was that the theoretically fastest algorithm for DCT (FastDCT) was not the best performing algorithm on GPU, in contrast to for Cell. Further, the data granularity chosen had a significant impact on the performance, which further complicates portability. Deciding what the correct code and data granularity should be is very hard to do without trial and error, since it differs with architecture and even different models of the same architecture (microarchitectures).

2.3 Asymmetric shared memory architectures

2.3.1 Overview

Asymmetric shared memory architectures have processing elements with varying features that can all access a shared memory. The cores may vary by instruction set, clock frequency, capabilities or other features and are interconnected in a number of ways. Having shared memory, even though the processing elements varies, enables programmers to directly access data on all cores allowing for convenient inter process communication. There are several good reasons to have asymmetric cores in a modern architecture. For example, by having specialized capabilities enabling efficient processing of particular workloads such as network traffic, vector mathematics or cryptographic functions, performance can be increased considerably. Another reason is to avoid duplicating unneeded capabilities, e.g., vector instructions may not be needed on processing elements that only work on scalar data. Further, clock frequencies can be scaled to match resource requirements and power budgets.

One example of asymmetric shared memory architectures is the IXP architecture, originally developed by Intel and now owned by Netronome, designed for high throughput, low latency network processing. The IXP2400 has 8 specialized network processors called μ engines, coupled with an ARM core running a traditional operating system. Several memory types are available including SRAM, DRAM and CAM, all with different properties, but available for direct use by the different processing elements. More details of this is given in section 2.3.2, where we look at a particular application of network processing on this architecture. Asymmetric cores are particularly interesting in the domain of low power devices since unneeded capabilities can be disabled on some of the cores to reduce power consumption. One example of this is the Tegra 3 processor from Nvidia with what they refer to as Variable Symmetric Multiprocessing [40]. In this processor, the asymmetry is in the manufacturing technology of the cores, i.e., one or more cores use fast switching gates allowing high throughput (and clock frequency), while others use power efficient gates limiting clock frequency, with the benefit of low power usage when little computational resources are required. The Tegra 3 processor has four fast cores, and one companion (slow) core, and in this implementation, the switch from one to four or vice versa is transparent to the software.

2.3.2 Case study: Video protocol translation on IXP

Streaming services are today almost everywhere available. Major newspapers and TV stations make on-demand and live audio/video (A/V) content available, video-on-demand services are becoming common and even personal media are frequently streamed using services like podcasting or uploading to streaming sites such as YouTube.

The discussion about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability and jitter than could not be offered by TCP. Today, this approach is impeded with middleboxes in Internet service providers (ISPs), by firewalls in access networks and on end-systems. ISPs reject UDP because it is not fair against TCP traffic, many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP-streaming, which delivers streaming media over TCP. The disadvantage is that the end-user can experience playback hiccups and quality reductions because of the probing behavior of TCP, leading to oscillating throughput and slow recovery of the packet rate. A sender that uses UDP would, in contrast to this, be able to maintain a desired constant sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that performs a transparent protocol translation. This is similar to the use of proxy caching that ISPs employ to reduce their bandwidth, and we do in fact aim at a combined solution. There are, however, too many different sources for adaptive streaming media that end-users can retrieve data from to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves a high perceived quality to end-users. Our prototype proxy is implemented on an Intel IXP2400 network processor and enables the server to use UDP at the server side and TCP at the client side.

In this section, we describe our IXP2400 implementation of a dynamic transport protocol translator. Preliminary tests comparing HTTP video streaming from a web-server and RTSP/RTP-streaming from the komssys video server show that, in case of some loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at play out rate while the TCP stream oscillates heavily. This work was first presented at ACM Multimedia 2007 [41], and later demonstrated live at ACM NOSSDAV 2008 [42].

Related Work

Proxy servers have been used for improved delivery of streaming media in numerous earlier works. Their tasks include caching, multicast, filtering, transcoding, traffic shaping and prioritizing. In this work, we want to draw attention to issues that occur when a proxy is used to translate transport protocols in such a way that TCP-friendly transports mechanisms can be used in backbone networks and TCP can be used in access networks to deliver streaming video through firewalls. Krasic et al. argue that the most natural choice for TCP-friendly traffic is using TCP itself [43]. While we agree in principle, their priority progress streaming approach requires a large amount of buffering to hide TCP throughput variations. In particular, this smoothing buffer is required to hide the rate-halving and recovery time in TCP's normal approach of probing for bandwidth that grows proportionally with the round-trip time. To avoid this large buffering requirement at the proxy, we would prefer an approach that maintains a more stable packet rate at the original sender. The survey presented in [44] shows that TFRC is a reasonably good representative of the TCP-friendly mechanisms for unicast communication. Therefore, we have chosen this mechanism for the following investigation.

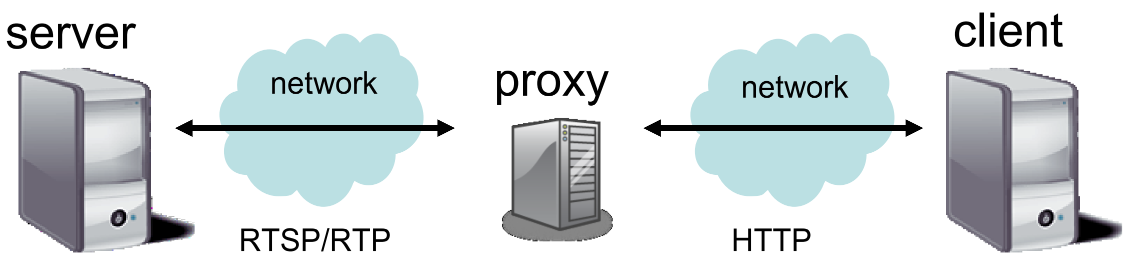


Figure 2.18: System overview

With respect to the protocol translation that we describe here, we do not know of much existing work, but the idea is similar to the multicast-to-unicast translation [45]. We have also seen voice-over-IP proxies translating between UDP and TCP. In these examples, a packet is translated from one type to another to match the various parts of the system, and we here look at how such an operation performs in the media streaming scenario.

Translating proxy

An overview of our protocol translating proxy is shown in figure 2.18. The client and server communicates by the proxy, which transparently translates between HTTP and RTSP/RTP. Both peers are unaware of each other.

The steps and phases of a streaming session follows: The client sets up a HTTP streaming session by initiating a TCP connection to the server; all packets are intercepted by the proxy, and modified before passing it on to the streaming server. The proxy also forwards the TCP 3-way handshake between client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connections. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file using RTP/UDP. The UDP packets are translated by the proxy as part of the HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection

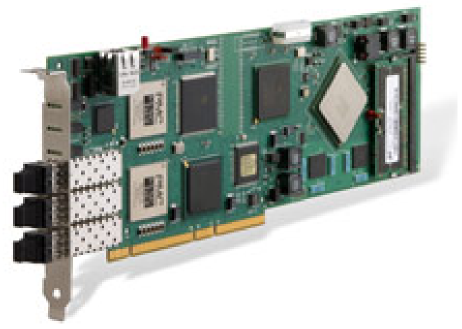


Figure 2.19: ENP-2611 from RadiSys with an IXP2400 Network Processor. The ENP-2611 is a PCI-X card working independently of the host computer. It allows DMA transfers to and from the host computer using the PCI bus.

is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

Implementation

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [46]. The chipset is a second generation, highly programmable network processor (NPU) and is designed to handle a wide range of access, edge and core network applications. The basic elements include a 600 MHz XScale (ARM compatible) core running Linux or VxWorks, eight 600 MHz special packet processors called microengines (μEngines), several types of memory and different controllers and buses.

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

The major functional blocks of the IXP2400 architecture are shown in figure 2.20. The μEngines are grouped together in clusters of four, which can communicate by next-neighbour registers internally. In normal configurations, two μEngines are reserved for low-level network receive and transmit functions, leaving six μEngines available for application usage.

The SDK includes a specialized C compiler for the MicroC language where the μEngines

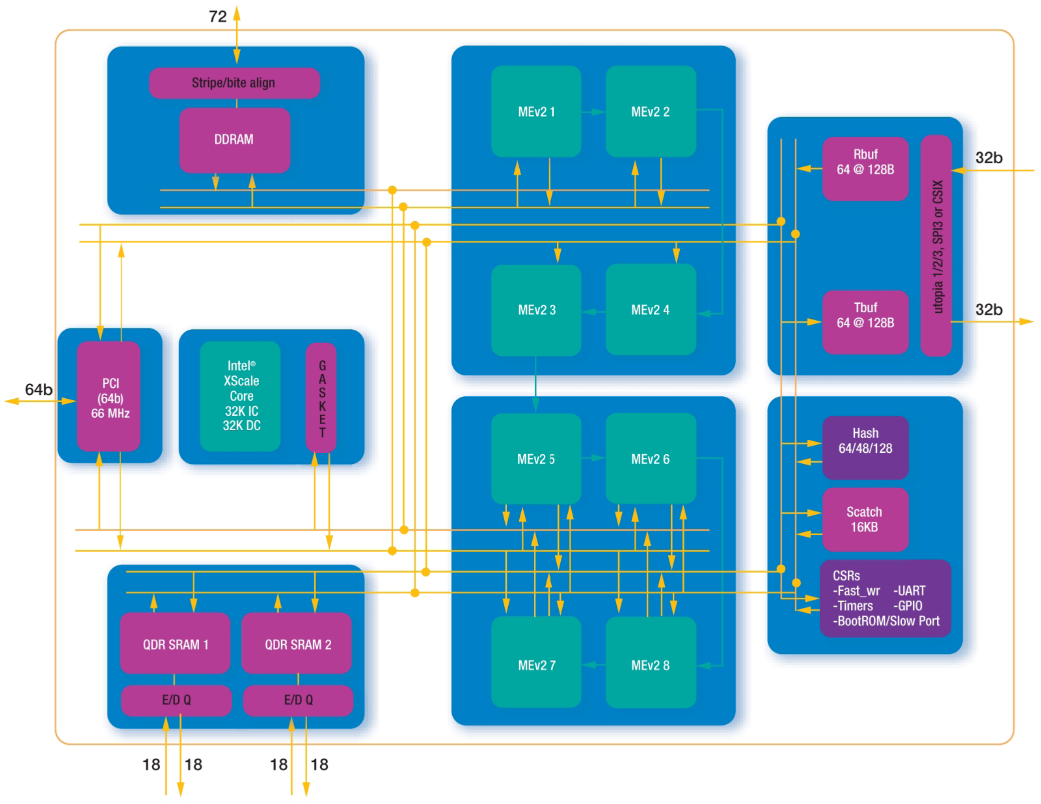


Figure 2.20: IXP2400 architecture overview. The figure is from an Intel product brief on IXP2400 [1]

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

The transport protocol translation operation[2] is shown in figure 2.21. The protocol translation proxy uses the XScale core and one μEngine application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. RTSP and HTTP packets are enqueued for processing on

[2]Our proxy also performs proxying of normal RTSP sessions and transparent load balancing between streaming servers, but this is outside of the scope of this work. We also have unused resources (μEngines) enabling more functionality.

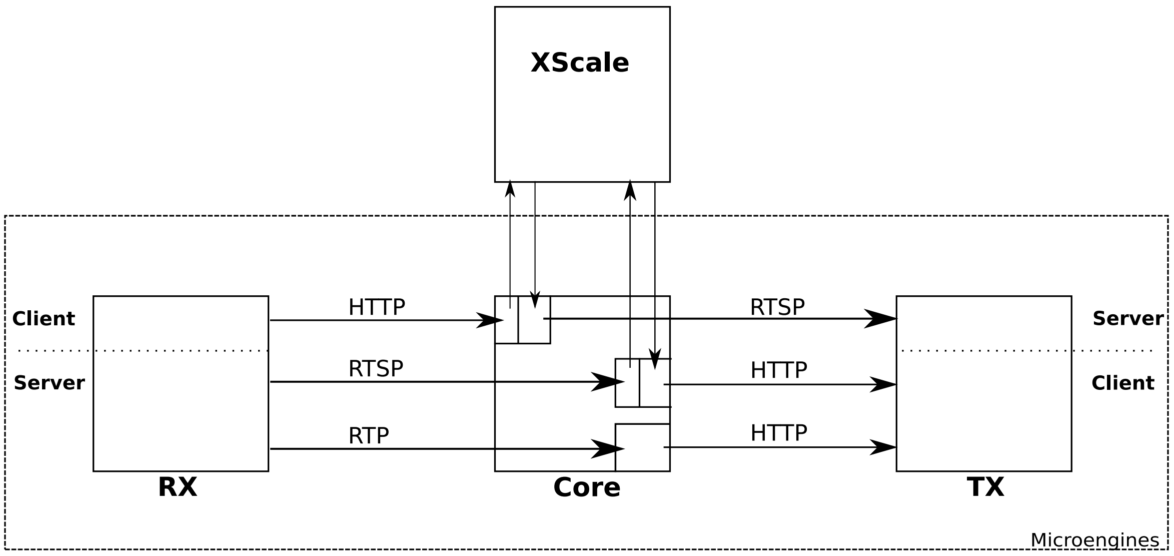


Figure 2.21: Protocol translator packet flow on the IXP2400

the XScale core (control path) while the handling of RTP packets is performed on the μEngine (fast path). TCP acknowledgements with zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after the initialization, all video data is processed (translated and forwarded) by the μEngines. The proxy supports a partial TCP/IP implementation, covering basic features. This is done to save both time and resources on the proxy.

To be fair to competing TCP streams, we implemented congestion control for the client loss experiment. TFRC [47] computation is used to determine the bandwidth available for streaming from the server. TFRC is a specification for best effort flows competing for bandwidth, designed to be reasonable fair to other TCP flows. The outgoing bandwidth is limited by the following formula:

$$X = \frac{s}{R * \sqrt{2 * b * \frac{p}{3}} + (t_{RTO} * 3 * \sqrt{3 * b * \frac{p}{8}} * p * (1 + 32 * p^2))}$$

where X is the transmit rate in bytes per second, s is the packet size in bytes, R is the RTT in seconds, b is the number of packets ACKed by a single TCP acknowledgment, p is the loss event rate (0-1.0), and t_{RTO} is the TCP retransmission timeout. The formula is calculated on a μEngine using fixed point arithmetic. Packets arriving at a rate exceeding the TFRC calculated threshold are dropped.

We are aware that this kind of dropping has different effects on the user-perceived quality than sender-side adaptation. We have only made preliminary investigations on the matter and leave it for future work. In that investigation, we will also consider the effect of buffering for at most 1 RTT.

Network Experiments and Results

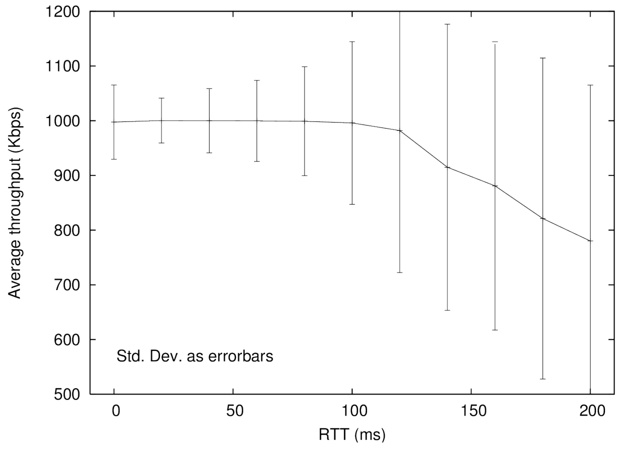
We investigated the performance of our protocol translation proxy compared to plain HTTP-streaming in two different settings. In the first experiment, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected proxy and client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0 - 200 ms) and the packet drop rate (0 - 1 %). We used a web-server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy whereas packets belonging to sessions initiated by connection made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump. Only the server-proxy loss experiments are reprinted here, and for the full evaluation consult the paper [41].

The results from the test where we introduced loss and delay between server and proxy are shown in figure 2.22. The plot show that our proxy that translates transparently from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase.

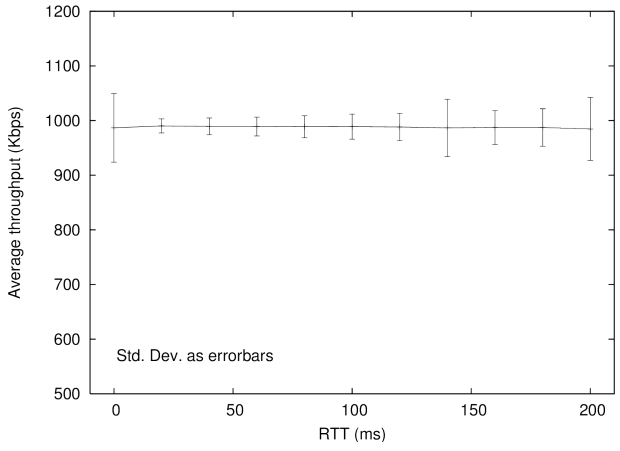
Discussion

Even though our proxy seems to give better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packet (and thus old data if the client does not buffer), the proxy fills the new packet with new updated data from the server. This means that the client in our prototype does not receive all data, and some artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed play out. This can cause video artifacts, depending on the codec used. However, this problem can easily be reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how it should be implemented. For this study, we have chosen the IXP2400 platform as we earlier have explored the offloading capabilities



(a) HTTP streaming



(b) Protocol translation

Figure 2.22: Achieved bandwidth varying drop rate and link latency with 1% server-proxy loss

of such programmable network processors. Using such an architecture, the network processor is suited for many similar operations, and the host computer could manage the caching and persistent storage of highly popular data served from the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding at the cost of limited scalability and a potential higher latency.

Summary

Both TCP and UDP have their strengths and weaknesses. In this case study, we used a proxy that performed transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enabled the server to use UDP on the server side and TCP on the client side. The server gained scalability by not having to deal with TCP processing. On the client side, the TCP stream was not discarded and passed through firewalls. The experimental results show that our protocol transparent proxy achieved translation and delivers smoother streaming than HTTP-streaming.

2.3.3 Implications

The shared memory of these architectures make them easy to program, but at the potential cost of unnecessary memory transfers by prefetching to local caches. Further, cache coherency protocols themselves consume resources and may inhibit scalability. Heterogeneous cores may require several versions of the same code to be written depending on what processing element that executes it. In terms of multimedia processing, shared memory is beneficial with regard to programmability, but may be unnecessary when the data access patterns are highly local. We saw in section 2.3.2 how using shared memory provided flexible data transfers between the cores in combination with a hardware assisted FIFO for signaling. However, the asymmetric cores also require specialized tools, compilers and a complicated development environment forcing software to be written from scratch for the architecture. The advantage gain of such an architecture thus depends on the application; the highly optimized hardware capabilities and low latency benefits may outweigh the programming complexity when they are needed. It is fairly obvious that writing the protocol translating proxy software on a standard Linux box would require significantly less effort, but performing at line speed 1 Gbps and beyond is not something easily accomplished on a standard PC until recently. As such, the asymmetric architectures provide great flexibility and performance at the cost of portability of the application.

2.4 Portability, scalability and performance

What we see in common for all these architectures is that code partitioning, code placement and data locality all have huge effect on the achieved performance. The implication is that major architecture adaptations are required at the cost of portability. The typical scenario is a static schedule made with consideration to the architecture, e.g., tied to a specific GPU generation or CPU vector extensions instruction set, requiring duplication of effort to run on other hardware. A static schedule and code partition works fine in a dedicated processing scenario where one machine is set to perform one particular task continuously. However, if we change the workload (e.g., higher image resolution or a new stage in a pipeline), the static schedule may have to be remade, and depending on how the programs are structured, require a significant amount of work. Similarly, if the available computational resources change, e.g., by running on different (but compatible) hardware or by sharing resources with other applications, the static schedule may fail as well. In summary, we have seen that using different architectures for processing multimedia workloads can provide significant performance, but the cost of portability and scalability is high, requiring significant architecture adaptations to perform, or in some cases a complete rewrite of the workload.

The biggest challenge we saw was the very different memory layouts in the various architectures. This was even a problem with various implementations of the traditional x86 architecture resulting in different programs performing better depending on if it was running on an AMD or Intel processor - even though the instruction sets are identical. Much larger differences were observed when running on more exotic hardware such as the Cell Broadband Engine or Graphic Processors, were data partitioning in suitable chunks for processing played a huge role in the efficiency. In other words, having the relevant data *ready to go* when needed is imperative, and this is true whether we have a explicitly managed cache such as the local storage on Cell, or advanced hardware prefetching algorithms such as on Intel's Sandy Bridge microarchitecture.

By recognizing the limitations of the architectures, we will unify the differences in the next chapter, and propose a system for generalizing multimedia workloads in such a way that they can easily be ported between heterogeneous architectures. Further, such a system can help provide elasticity to scale resource usage to workload requirements and help various workloads share computational resources.

2.5 Summary

In this chapter, we have looked at case studies showcasing architecture considerations when running multimedia workloads on modern processors. Obtaining high performance requires detailed architecture knowledge and adaptations to the target architecture and severely limits portability of stand alone workloads. As such, using a framework that can provide dynamic scheduling for executing these types of workloads may provide better portability and dynamic resource usage.

Chapter 3

The P2G Framework

Inspired by the results of the previous chapter, we see a large potential for creating an abstraction for the low level architecture in such a way that we can retain the performance of hand-tuned multimedia applications while simultaneously providing portability to heterogeneous architectures. We found no existing system that provides such support, and to achieve this, we developed the P2G framework, an experimental platform for elastic execution and runtime scheduling of multimedia workloads, intrinsically designed with the following basic functions:

Abstractions suitable for multimedia workloads. Although multimedia workloads come in many forms, we pursue a suitable model for expressing workloads that transform dense data, typically expressed as matrices or similar structures. This is very different from MapReduce and other batch data processing models used in many elastic and distributed frameworks today.

Runtime support for running on heterogeneous architectures. Heterogeneous architectures present challenges otherwise not encountered in traditional computing. These include mapping the correct workloads to computational resources, but also dealing with multiple binary interfaces and instruction sets.

System elasticity allows the system to adapt to varying external conditions (resources), meaning that when available computational resources fluctuate, the runtime should continuously optimize the performance to match the available resources. System elasticity can be caused by sharing resources with other applications in a virtual machine, or executing multiple independent workloads executed on the system concurrently.

Workload elasticity allows the workload's computational requirements to change during runtime and the system should adapt to maximize the performance. Examples include dynamically adding or removing input streams (such as cameras), or modifying the pipeline while running. This is similar to system elasticity, but in contrast to the external events in system elasticity, the cause of change is internal.

Intrinsic soft real-time capabilities. Exposing deadlines directly in the framework allows the runtime to deal with deadline misses in a graceful manner, including mitigation techniques and down-prioritization of less important tasks.

Many frameworks and systems exist for elastic execution of batch workloads, but to our knowledge, no framework exists designed with these properties in mind. As such, we developed the P2G framework to experiment and evaluate whether these properties are achievable from a system's perspective.

The P2G project was developed in a collaboration of several PhD and Master students, and has many aspects that are not fully evaluated in this thesis, including support for distributed execution. In this chapter, we present the background and motivation for creating the framework, as well as design considerations and related work. In the last part of the chapter (section 3.5), we explain our prototype implementation and an evaluation of the prototype. The P2G project was first presented at IEEE SRMPDS, a workshop co-located with ICPP 2011 [48], and the prototype was later demonstrated live at ACM Multimedia 2011 [49] and Eurosys 2011 [50].

3.1 Related Work

A lot of research has been dedicated to addressing the challenges introduced by parallel and distributed programming. This has led to the development of a number of tools, programming languages and frameworks to ease the development effort.

For example, several solutions have emerged for simplifying distributed processing of large quantities of data. We have already mentioned Google's MapReduce [12] and Microsoft's Dryad [51]. In addition, there are IBM's System S and the accompanying programming language SPADE [52]. Yahoo has also implemented a programming language with their PigLatin language [53]. Other notable systems that provide increased language support are Cosmos [54], Scope [55], CIEL [56], SNAPPLE [57] and DryadLINQ [58]. These high-level languages provide easy abstractions for the developers in an environment where mistakes are hard to correct.

Dryad, Cosmos and System S have many properties in common. They all use directed graphs to model computations and execute them on a cluster. System S also supports cycles in graphs, while Dryad supports non-deterministic constructs. However, not much is known about these systems, since no open implementations are freely available. MapReduce on the other hand has become one of the most-cited paradigms for expressing parallel computations. While Dryad and System S use a task-parallel model, MapReduce uses a data-parallel model based on keys and values. There are several implementations of MapReduce for clusters [11], multi-core [59], the Cell BE architecture [60], and also for GPUs [61]. Map-Reduce-Merge [62] adds a merge step to process data relationships among heterogeneous data sets efficiently, operations not directly supported by the original MapReduce model. In Oivos [63], the same issues are addressed, but in addition, this system provides a more expressive, declarative programming model. Finally, reducing the layering overhead of software running on top of MapReduce is the goal of Cogset [64] where the processing architecture is changed to increase performance.

An inherent limitation of MapReduce, Dryad and Cosmos is their inability to model iterative algorithms. In addition, the rigid MapReduce semantics does not map well to arbitrary problems [62], which may lead to unnaturally expressed solutions and decreased performance [65]. The limited support for iterative algorithms has been mitigated in HaLoop [66], a fork of Hadoop optimized for batch processing of iterative algorithms where data is kept local for future iterations of the MapReduce steps. Furthermore, the programming model of MapReduce is designed for batch processing huge datasets, and not well suited for multimedia algorithms.

Kahn Process Network-based (KPN) frameworks are one alternative to batch processing frameworks. KPNs support arbitrary communication graphs with cycles and are deterministic. However, in practice, very few general-purpose KPN runtime implementations exist. Known implementations include the Sesame project [67], the process network framework [68], YAPI [69] and our own Nornir [70]. These frameworks have several benefits, but for application developers, the KPN model poses some challenges, particularly in a distributed scenario. To mention some issues, a distributed version of a KPN implementation requires distributed deadlock detection and a developer must specify communication channels between the processes manually.

An alternative framework based on a process network paradigm is StreamIt [71], which comprises a language and a runtime system for simplifying the implementation of stream programs described by a graph that consists of computational blocks (filters) with a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [72]. The compiler produces code that can make use of multiple machines or CPUs, whose number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications. Multimedia applications have strict performance requirements, and real-time awareness is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resources change. All of the existing frameworks have some shortcomings that are difficult to address, and the traditional batch processing frameworks come up short in our multimedia scenario. Next, we present our ideas for a new framework for distributed real-time multimedia processing.

3.2 Basic idea

The idea of P2G was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that many of them sacrifice either data or task parallelism, two orthogonal dimensions for expressing parallelism. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they limit the ability to express parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but they also restrict runtime processing to the domain of data parallelism [73]. Functional languages such as Erlang [74] and Haskell [75] and the event-based Specification and Description Language (SDL) [76], map well to task parallelism. However, functional languages fail because the step from math formulas to recursive formulation is too hard for most programmers and using global state is excessively hard, and thus not very suited for multimedia workloads.

In our prior work with the system Nornir [70], we improved on many of the shortcomings of the traditional batch processing frameworks, like MapReduce [12] and Dryad [51]. Nornir was not designed for multimedia processing specifically, and implements instead the general concept of KPN. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles/iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs. However, Nornir is task-parallel, and the programmer must add data-parallelism explicitly. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, connecting exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations, however, queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

With the P2G approach presented here, we build on the knowledge gained from developing Nornir and address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature for processing multimedia workloads includes automatically combined task and data parallelism. Intra-frame prediction in H.264 AVC, for example, introduces many dependencies between sub-blocks of a frame, and together with other overlapping processing stages, these operations have a high potential for benefiting from both types of parallelism. We demonstrated the potential in earlier work with Nornir and showed great parallelization potential in processing arbitrary dependency graphs.

Multimedia algorithms, being cyclic in nature, exhibit many pipeline-parallel opportunities. Exploiting them is hard because intrinsic knowledge of fine-grained dependencies is required, and structuring programs in such a way that pipeline parallelism can be used is difficult. Thies et al. [77] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses assuming that the behaviour is stable. They evaluated their system on multimedia algorithms and gained significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, and this enables the runtime to automatically detect and take full advantage of all parallel opportunities without manual intervention.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations to and from memory. The source of this non-deterministic behavior can be removed by adopting strict write-once semantics for writing to memory [78]. Languages that take advantage of this concept, also known as single assignment, include Erlang [74] and Haskell [75]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted in P2G. While write-once semantics are convenient for a scheduler's dependency analysis by allowing the dependency tree to be expanded into a DAG, it is not straight-forward to think about multimedia algorithms in functional terms such as the language of Erlang or Haskell. Therefore we invented a new and different approach for P2G.

Multimedia algorithms are often formulated in terms of cycles of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [79] and SIFT [80] pipelines. Both are also examples of algorithms whose consecutive steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of arrays without loosing write-once semantics.

3.2.1 Flexible partitioning using fields

The P2G system is designed to be flexible, both for the workload writer and from a runtime perspective. To achieve this, we provide flexible partitions of data, termed *field*, to abstract data access. A field allows workload writers to access data elements in a similar way as traditional multi-dimensional arrays, but with an important restriction: Data elements can only be written once.

This simple restriction forms the core of the P2G idea. By supporting the familiar concept of multi-dimensional arrays in combination with sequential code blocks performing work on the fields, we provide programmers with a simple to use interface for workloads. At the same time, the write-once semantics is very powerful; it allows the runtime system to analyze code and data dependencies completely at the finest level of granularity without manual specification. As such, by using fields to describe a workload instead of traditional arrays, the runtime can analyze exactly what data is required to perform the next step and also determine the operations that can be executed in parallel without side effects. The fields allow the system to perform flexible partitioning of data at runtime. Further, code dependencies are also exposed when using the fields because code can only run when the field elements it depends upon have been written to. Since this can only happen once, the execution of a code block can start as soon as the field is written to. Note that since the values in a field will never change once set, it is safe to make a copy of the content and run the code in isolation, e.g. on a different core or another machine.

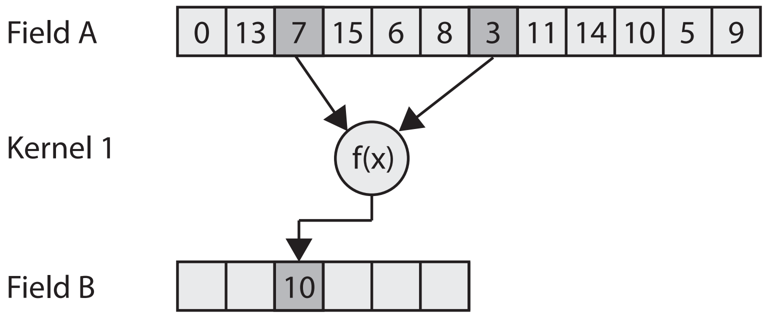


Figure 3.1: One kernel reading data from Field A, applies a function and writes the result to Field B. Note that most elements of Field B is empty, and cannot be read by another kernel until written to, resulting in an implicit dependency.

We call these code blocks kernels.

Kernels have strict dependencies that must be fully satisfied before execution. The dependencies are specified as *fetch* operations on a number of field elements. As a result of execution, the kernel outputs a number of elements in one or more fields using *store* operations, which in turn satisfy dependencies for other kernels. See figure 3.1 for an example on how this is done. The kernels and fields are central concepts of the P2G idea, and in combination they give the runtime full knowledge of data and code dependencies without manually specifying these in an additional meta language. This allows the runtime to make side-effect free decisions on where a particular kernel should be run, when it is able to run, and what data must have been produced by earlier kernels for it to be able to run. The write-once requirement of fields ensures deterministic output regardless of execution order and where a kernel executes. The dependency analysis can be performed at runtime, at compile time or in a combination and allows the runtime great flexibility on the way in which it executes a workload.

In its basic form, a kernel describes the transformation of one or more multi-dimensional input fields to output fields. The fields provide virtual access to data regardless of where it actually resides, e.g., the the data can be in memory, on disk or on a remote machine. Reading or writing data from or to fields forms implicit dependencies, which are used to control execution of the workload. The fields can be of any number of dimensions, and we do not restrict the access pattern. This allows us to read or write in arbitrary order, which forms dependencies on the exact data needed. An example is shown in figure 3.2, where a stencil kernel accesses a 2D field. This notion of arbitrary access is inspired by Cray's Chapel language [81], which allows complex reading and writing of multidimensional arrays. Chapel, however, does not have the assumption of write-once semantics, which makes it a very different language and framework than P2G.

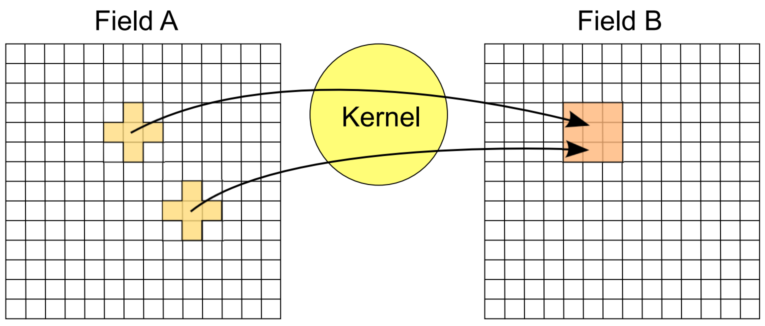


Figure 3.2: 2D-Field A is input to our stencil kernel, resulting in Field B. Dependencies are resolved as needed, so not all data in Field A need to be stored for the kernel to run, allowing side effect free execution.

3.2.2 Write-once semantics

The *fields* have strict write-once semantics, implying that a particular data point will never change. If a kernel tries to read an unwritten data point, the dependency analyzer will stall this kernel until the required data is written. This is in contrast to a traditional blocking read, which stalls at the time of read - P2G does not even try to run the kernels when dependencies are not satisfied. The write-once policy further implies that two kernels will never write to the same location in memory. The write-once policy has several benefits, easily exploited in our framework: First, the execution is deterministic regardless of execution order, allowing the scheduler to run kernels in any order (as long as the dependencies are satisfied). This property is common in functional programming languages, and allows us to avoid sorting execution order of kernel instances. Second, the property of write-once semantics allows side-effect free parallelism. Since kernels running in parallel will never write to the same location in memory, they can run in total isolation without synchronization, allowing us to move a kernel to another core or another machine entirely without considering parallel synchronization other than through fields. This enables convenient execution on exclusive memory architectures as we saw in section 2.2, letting the scheduler know that all dependencies are satisfied before execution. Third, since we have fine-grained dependencies on the smallest units of data, we can start the execution of kernels that read from a field produced by another kernel before all elements in this field are written.

Using write-once semantics of the fields, we try to combine the side-effect free and efficient execution of functional programming languages, with the ease of programming multimedia workloads in imperative programming languages. We see the kernels as functional blocks that transform one field into another. The write-once semantics of the fields forms an implicit dependency graph between all elements of the fields. The dependencies can be further expanded into a directed acyclic graph at runtime and analyzed in many ways before execution. How this is done is an implementation question, and we describe this further in section 3.5 covering our prototype implementation.

3.2.3 Kernels

A kernel is the other basic building block in the P2G framework. Kernels perform fetch and store operations on fields, and apply a transformation of the input data to the output. As such, kernels can be seen as mathematical functions, and we have designed the P2G language to support traditional imperative code to do the transformation. This transformation code can be written in any language as long as appropriate bindings are provided. In the prototype covered later, we use a C++ binding for writing kernel code.

A kernel describes the dependencies between fields and their elements and is written using relative terms, e.g., it takes two subsequent elements in a 1D-field as input and outputs one element to another 1D-field. A specific instance of this kernel, taking a specific pair of field elements as inputs and another field element as output is called a *kernel instance*. Splitting a field into smaller units for use in a single kernel instance is called *field slicing* and is a concept that we adopted from the Python programming language [82]. Since kernel instances rely only on fields, they can be executed in arbitrary order once the dependencies are satisfied. Multiple versions of a kernel can be defined providing code for different architectures or implementations, allowing a GPU version of a specific kernel to co-exist with an x86 version. Note that after an instance of a kernel version has run, another version of the same instance cannot execute since they will store the result to the same field elements. As such, the dependencies for running a specific kernel instance are only satisfied once.

3.2.4 Aging

Having the kernels and fields as defined above, we can apply transformations on the input data in a deterministic manner. However, to support cyclic workloads, which were a goal from the beginning, we need a mechanism for handling continuous data streams. To support this, we introduce the concept of *aging fields*. The age adds a new dimension to all fields, allowing us to add a new *age* to a field when a new round in a cycle starts. The concept of aging allows us to keep strict write-once semantics while at the same time enabling us to support arbitrarily long cycles.

3.2.5 P2G Goals

Together, the these four basic ideas form the design of the P2G framework:

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *write-once semantics* for the fields to achieve deterministic behavior and side-effect free parallelism.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

Within the boundaries of these basic ideas, P2G should be easily accessible for programmers who only need to write isolated, sequential pieces of code embedded in kernel definitions. The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to fetch slices of a field in as fine granularity as possible, supporting data parallelism. P2G is designed to be language independent, although, we have defined a C-like language that captures many of P2G's central concepts. As such, the P2G language is inspired by many existing languages. In fact, Cray's Chapel [81] language antedates many of P2G's kernel language features in a more complete manner. P2G adds, however, write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements that formulate data dependencies between implicitly instantiated kernels, which are (currently) written in C/C++. The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations.

In summary, we have opted for an idea that allows programmers to focus on data transformations in a sequential manner, while simultaneously providing enough information for dynamically adapting the data and task parallelization. As an end result of our considerations, P2G's fields look mostly like global multi-dimensional arrays in C, although their representation in memory may deviate, i.e., they need not be placed contiguously in the memory of a single node, can be distributed across multiple machines or may not exist in memory at all. Although this looks contrary to our message-based KPN approach used in Nornir [83], it maps well when slices of fields are interpreted as messages and the run-queues of worker threads as KPN channels. An

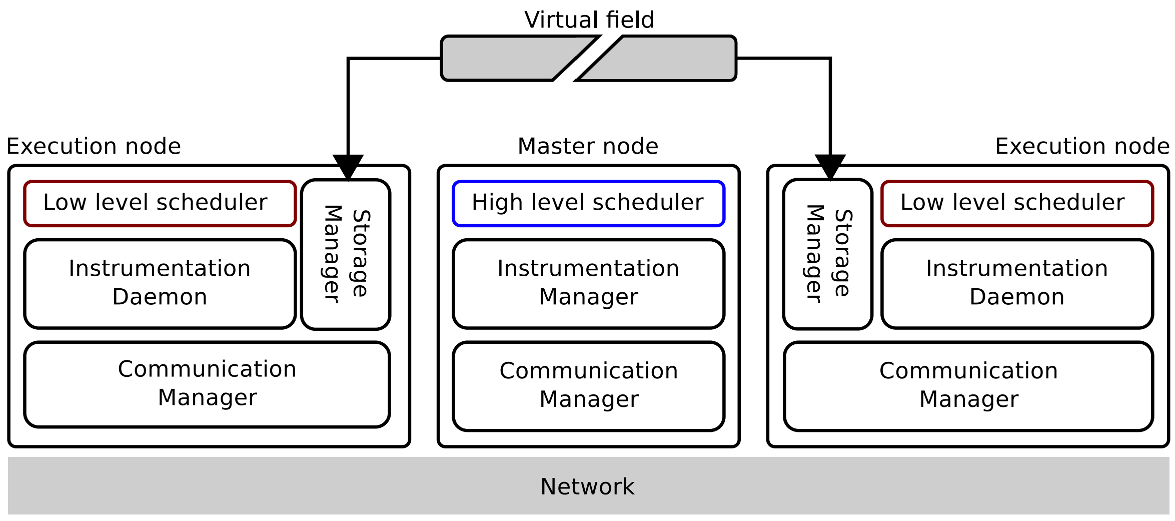


Figure 3.3: Overview of the architecture in a distributed P2G system.

obvious difference is that fields can be read as often as necessary, but this can be interpreted as an invisible copying KPN process.

3.3 Architecture

As shown in figure 3.3, the P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (a graph of multi-core and single-core CPUs and GPUs, connected by various kinds of busses and other networks) to the master node. Finally, the master node compiles this information into a global topology of available resources. As such, the global topology can change during runtime as execution nodes are dynamically added and removed to accommodate changes in the global load.

```
**Input**: $m\_data[0 : n-1]$
**Output**: $p\_data[0 : n-1]$
**while** *true* **do**
    mul2 kernel:
    **for** $i \leftarrow 0$ **to** $n$ **do**
        $p\_data[i] \leftarrow m\_data[i] * 2$
    **end**
    plus5 kernel:
    **for** $i \leftarrow 0$ **to** $n$ **do**
        $m\_data[i] \leftarrow p\_data[i] + 5$
    **end**
**end**
```

Algorithm 1: Mul-Sum example workload

The P2G system is designed to handle distributed computing as well as processing on a single machine. To optimize throughput, we propose using a two-level scheduling approach: On the master node, we have a high-level scheduler, and on the execution node(s), we use a low-level scheduler. The operating system scheduler is decoupled by pinning thread affinity, hence the P2G low-level schedulers can anticipate and mitigate cache affinity issues, whose importance we saw in section 2.1.2. Further, an instrumentation daemon provides detailed execution performance parameters to the scheduler in realtime, allowing the scheduler to make appropriate decisions based on performance data. The high-level scheduler is only used for distributed computing, and is responsible for assigning and distributing work units to the different nodes. This idea of using a two level scheduling approach is not new. It has also been considered by Roh et al. [84], who have performed simulations of parallel scheduling decisions for instruction sets of a functional language. Simple workloads are mapped to various simulated architectures, using a "merge-up" algorithm, which is equivalent to our low-level scheduler, and "merge-down" algorithm, which is equivalent to our high-level scheduler. These algorithms cluster instructions in such a way that parallelism is not limited. Their conclusion is that utilizing a merge-down strategy often is better. Although distributed execution in the P2G system is an interesting goal, we do not thoroughly cover the topic in this thesis. Details on how distributed computing support can be added to the P2G framework can be found in Paul Beskow's thesis [85], who co-authored the P2G design and focused on the distributed aspect of the framework.

3.4 Programming model

The program model of P2G allows flexible task, data and pipeline parallelism using kernels and fields as basic constructs. Throughout this section, we use a very simple example, which we refer to as the *mul-sum* workload, to demonstrate how the framework is designed. The mul-sum workload is a very simple workload that takes an array of integers as input. Every integer is first multiplied by two by the mul2 kernel and stored to an intermediate array. Elements of this array are in turn increased by 5 by the plus5 kernel, producing a new array. This array is used as input to the mul2 kernel again, and the workloads run continuously. Pseudocode of the mul-sum workload is listed as algorithm 1.

The mul-sum workload does not do very useful work, but it allows us do demonstrate various features of the P2G framework. P2G can derive an implicit dependency graph from the kernels and fields, more specifically from the access patterns on fields imposed by fetch and store state-

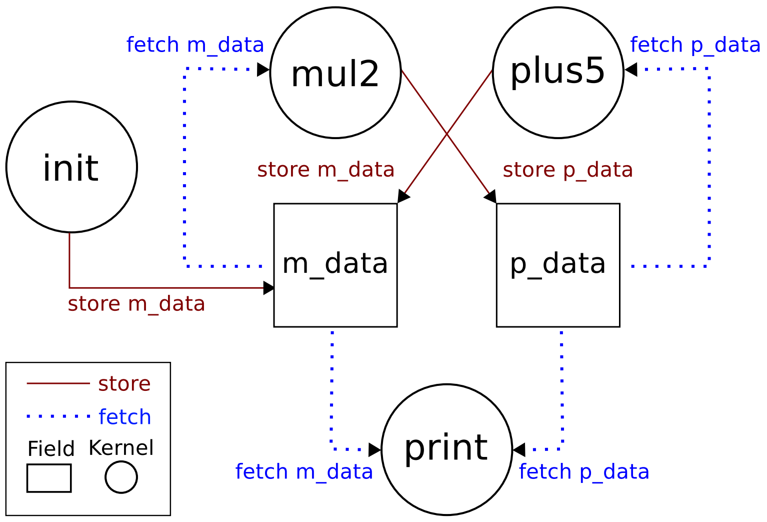


Figure 3.4: Implicit dependency graph of the mul-sum workload.

ments. An illustration of the implicit dependency graph as seen by the P2G scheduler is shown in figure 3.4. In addition to the mul2 and plus5 kernels, we have a kernel for initialization and a kernel that continuously prints the field values to screen. The implicit dependency graph determines the fields that are required for a kernel to run, and the fields that are produced as a result of execution. There is no termination condition for this workload, hence it will run indefinitely once started.

During runtime, the implicit dependency graph is expanded to form a dynamically created directed acyclic dependency graph (DC-DAG), as seen in figure 3.5. This expansion from a cyclic graph to a directed acyclic graph occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iteration. We have chosen to call each such unrolled loop an *Age*, as introduced in section 3.2.4. The low-level scheduler can then use the DC-DAG to combine tasks and data to reduce overhead introduced by P2G and to take advantage of specialized hardware, such as GPUs. It can then try different combinations of these low-level scheduling decisions to improve the throughput of the system.

We can see how this is accomplished in figure 3.5. When moving from *Age=1* to *Age=2*, we see how the low-level scheduler has made a decision to reduce data parallelism. In P2G, kernels fetch slices of data, and initially *mul2* was defined to work on each single field entry in parallel, but in *Age=2*, the low-level scheduler has decreased the granularity of the fetch statement to encompass the entire field. It could also have split the field in two, leading to two kernel instances of *mul2*, working on distinct sets of the field.

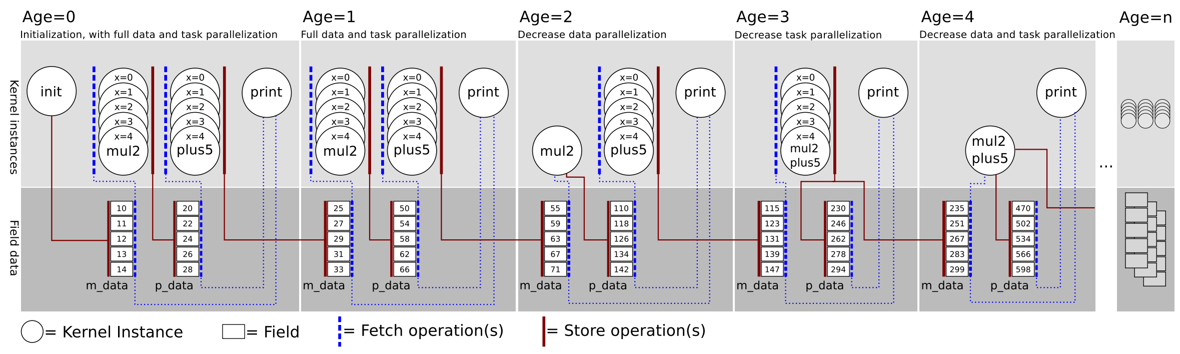


Figure 3.5: Dynamically created directed acyclic dependency graph (DC-DAG). The scheduler makes decisions to reduce task and data parallelization, which are reflected in the dynamically created graph for every age (iteration).

This granularity reduction is inspired by what Ian Foster introduced as Agglomeration in his book [86]. Agglomoration means combining tasks into larger tasks, and in this case we reduce data parallelism based on a scheduler decision. To reduce the overhead of executing tasks with small units of data, the runtime can combine them into larger units.

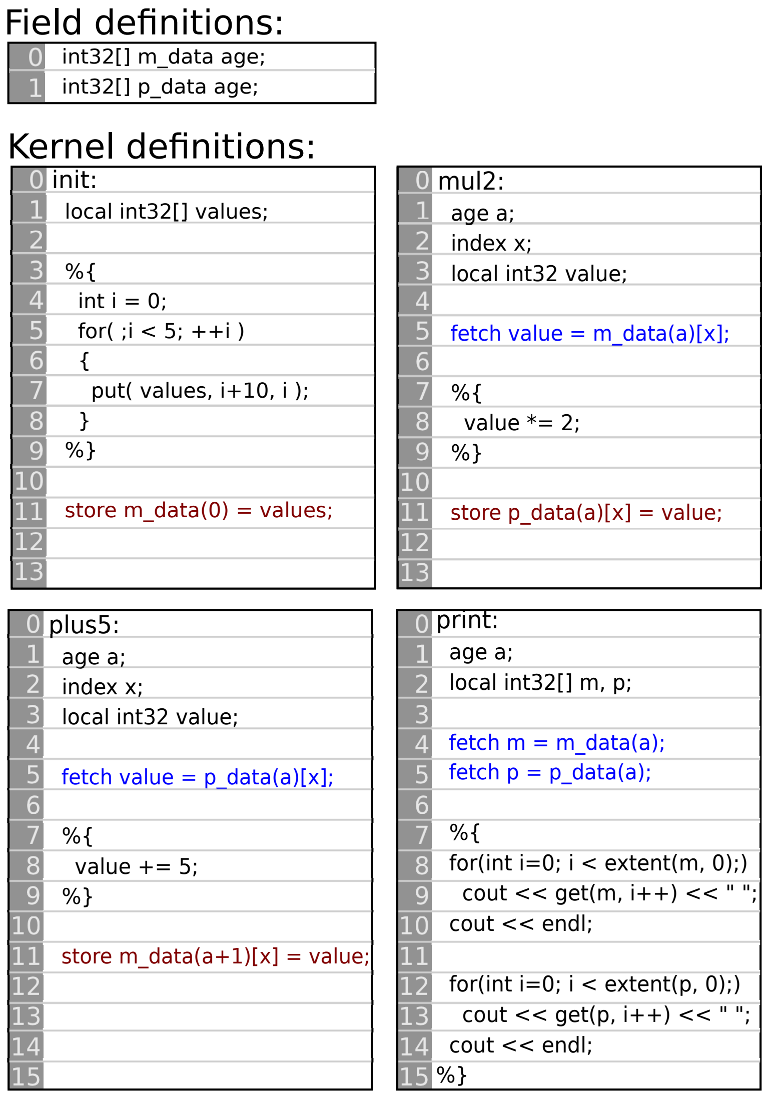
Moving from *Age=2* to *Age=3*, we see how the low-level scheduler has made a decision to decrease the task parallelism. This is possible because the *mul2* and *plus5* kernels effectively form a pipeline, information that is available from the implicit graph. By combining these two tasks, the individual store operations of the tasks are deferred until the data has been fully processed by each task. If the *print* kernel was not present, storing to the intermediate field *m_data* could be circumvented entirely.

Finally, moving from *Age=3* to *Age=4*, we can see how a decision to decrease both task and data parallelism has been taken. This renders this single kernel instance effectively into a classical *for-loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the data.

P2G makes such runtime adjustments dynamically to both data and task parallelism, based on the possibly oscillating resource availability and the reported performance monitoring.

3.4.1 Kernel language

From our experience with Nornir, we came to the realization that expressing workloads in a framework capable of supporting such complex graphs without a high-level language is a difficult task. We have therefore developed a *kernel language* to express relationships between kernels and fields. An implementation of the mul-sum workload is outlined in figure 3.6, with a C++ equivalent listed in figure 3.7.



Field definitions:

```
int32[] m_data age;
int32[] p_data age;
```

Kernel definitions:

```
init:
  local int32[] values;

  %{
    int i = 0;
    for( ;i < 5; ++i )
    {
      put( values, i+10, i );
    }
  %}

  store m_data(0) = values;


```

```
mul2:
  age a;
  index x;
  local int32 value;

  fetch value = m_data(a)[x];

  %{
    value *= 2;
  %}

  store p_data(a)[x] = value;


```

```
plus5:
  age a;
  index x;
  local int32 value;

  fetch value = p_data(a)[x];

  %{
    value += 5;
  %}

  store m_data(a+1)[x] = value;




```

```
print:
  age a;
  local int32[] m, p;

  fetch m = m_data(a);
  fetch p = p_data(a);

  %{
  for(int i=0; i < extent(m, 0);)
    cout << get(m, i++) << " ";
  cout << endl;

  for(int i=0; i < extent(p, 0);)
    cout << get(p, i++) << " ";
  cout << endl;
%}
```

Figure 3.6: Mul-Sum workload kernel and field definitions

```
void print( int *in, int num )
{
    for( int i = 0; i < num; ++i )
        std::cout << in[i] << "␣";
    std::cout << std::endl;
}

void mul2(int *in, int *out, int num)
{
    for (int i = 0; i < num; ++i)
        out[i] = in[i] * 2;
}

void plus5(int *in, int *out, int num)
{
    for (int i = 0; i < num; ++i)
        out[i] = in[i] + 5;
}

int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    int num = sizeof(m_data) / sizeof(m_data[0]);

    while( true )
    {
        mul2(m_data, p_data, num);

        print( m_data, num );
        print( p_data, num );

        plus5(p_data, m_data, num);
    }
    return 0;
}
```

Figure 3.7: C++ equivalent of mul/sum example

In the current version of our system, P2G is exposed to the developer through this *kernel language*. The language itself is not an integral part and can be replaced easily. However, it exposes several foundations of the P2G design. Most important are the kernel and field definitions, which describe the code and interaction patterns in P2G.

A kernel definition's primary purpose is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. As such, a field serves as an interaction point for kernel definitions, as can be seen in figure 3.4.

Fields in P2G have a number of properties, including a type and a dimensionality. Another property is, as mentioned above, *aging*, which allows kernels to be cyclic while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times, as long as the age increases for each store operation (as seen in figure 3.5). In essence, this adds a dimension to the field and makes it possible to accommodate iterative algorithms. Additionally, it is important to realize that fields are not connected to any single node, and can be fully localized or distributed across multiple execution nodes (as seen in figure 3.3).

In defining the interaction between kernels and fields, it is encouraged that the programmer expresses the finest possible granularity of kernel definitions, and, likewise, the most precise slices possible for the kernel within the field. This is encouraged because it provides the low-level scheduler more control over the granularity of task and data decomposition. Aided by instrumentation data, it can reduce scheduling overhead by combining several instances of a kernel that process different data, or several instances of different kernels that process data in sequence (as seen in figure 3.5). The scheduler makes its decisions based on the implicit dependency graph and instrumentation data.

An important aspect of multimedia workloads is the ability to express deadlines, e.g., where it does not make sense to encode a video frame if the playback time has moved past a certain point in the video-stream. Consequently, we have implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer: *timer t1*. This timer can then be polled, and updated from within a kernel definition, for example *t1+100ms* or *t1 = now*. Given a condition based on a deadline such as *t1+100ms*, a timeout can occur and an alternate code-path can be executed. Such an alternate code-path is executed by storing to a different field than in the primary path, leading to new dependencies and new behavior. Currently, we have basic support for expressing deadlines in the kernel language, but the semantics of these expressions require refinement, as their implications can be considerable.

3.4.2 Runtime

Following from the previous discussions, we can extent the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime, and the number of kernel instances executed in parallel for a given kernel definition depends on its fetch statements, known only at runtime.

To clarify, a kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements of the kernel definition. For example, looking at figures 3.5 and 3.6, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x* fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum possible *x* kernel instances simultaneously per age, giving *a***x mul2* kernel instances. As we have seen, this number can be decreased by the scheduler making *mul2* work over larger slices of data from *m_data*.

With P2G we support implicit resizing of fields. This can be witnessed by looking at the kernel definition of *print* in figure 3.6. Initially, the extents of *m_data* and *p_data* are not defined, but with each iteration of the *for*-loop in *init* the local field *values* is resized locally, leading to a resize of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields effected by this resize, such as *p_data*. Following the discussion from the previous paragraph, such an implicit resize can lead to additional kernel instances being dispatched.

It is worth noting that a kernel instance is only dispatched when all its dependencies are fulfilled, i.e., the data it fetches has been stored in the respective fields and elements. Looking at figures 3.5 and 3.6 again, we can see that *mul2* stores its result in *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, the kernel instance *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. To summarize, the *print* kernel instance working on *age=0* becomes runnable when all the elements of *m_data* and *p_data* for *age=0* have been stored. Once it has become runnable, it is dispatched and runs only once.

3.4.3 Discussion

In P2G, we encourage the programmer to describe the workload in as fine granularity as possible, both in the functional and data composition domains. The low-level scheduler has an understanding of both composition domains and deadlines. Given this information, the low-level scheduler can minimize overhead by combining functional components and slices of data by adapting to its available resources, be it local cores, or even GPU execution units. Scheduling problems have been researched since the dawn of computer science and there are many approaches on how the low-level scheduler can achieve this, including heuristics-, graph- and feedback-based scheduling techniques. We believe the feedback based scheduling is most promising for the low-level scheduler after observing the widely varying performance implications of scheduler decisions in chapter 2. Subsequently, we design our prototype to react to changes in measured performance. The runtime can estimate overhead and throughput to evaluate if tasks and data should be combined into larger units or executed in a different order. This should be a constant effort by the runtime to optimize performance, and since the best behaviour is unknown, the low-level scheduler is free to probe for better configurations and continuously evaluate performance at runtime.

Write-once semantics on fields incur a large penalty if implemented naively, both in terms of memory usage and data cache misses. However, as the fields are virtual and do not even have to reside in continuous memory, the compiler and runtime are free to optimize field usage. This includes re-using buffers for increased cache locality when old ages are no longer referenced, and garbage collecting old ages. The explicit programming model of P2G allows the system to anticipate what data is needed in the future, which can be used for further optimizations.

Given the complexity of multimedia workloads and the (potentially) heterogeneous resources available in a modern topology, and in many cases, no knowledge of the underlying capabilities of the resources (common in modern cloud services), mapping these complex multimedia workloads manually to the available resources becomes an increasingly difficult task. This is particularly the case where resource availability fluctuates, such as in modern virtual machine parks. With batch processing, where the workloads are frequently not associated with an intrinsic deadline, the task is solved by frameworks such as MapReduce and Dryad. However, processing continuous streams such as iterative multimedia algorithms in an elastic manner requires new frameworks; P2G is a step in that direction.

In the next section, we describe a prototype with some of the P2G ideas implemented in addition to workloads and benchmarks. The P2G prototype provides, among other tings, opportunities to experiment with scheduling, instrumentation, runtime optimizations, and heterogeneous support which we consider applicable in other frameworks as well.

3.5 Evaluation of the P2G framework

With the express goal of providing and abstraction of the underlying architecture from P2G workloads, we designed a language that explicitly exposes all parallel opportunities for iterative multimedia workloads and allows a compiler and runtime system to make informed decisions on parallel opportunities that are worth exploring. This shifts the performance tuning burden from the programmer (workload writer) to the compiler and runtime (collectively referred to as the framework or P2G). Given a workload defined in the kernel language, we will in this section discuss how to execute it, as well as some of the optimization opportunities available to the framework. We will present a working P2G prototype for multicore x86 and evaluate its characteristics.

3.5.1 Prototype Implementation

To verify the feasibility of the P2G framework presented in this thesis, we have implemented a prototype version for x86. The prototype consists of a compiler for the kernel language and a runtime that can execute P2G programs on multi-core linux machines.

Compiler To parse and run kernel language files, an interpreting runtime was first considered and discarded for both performance and practical reasons. Programs written for the P2G system are designed to be platform-independent and feature blocks of code written in C or C++. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as nVIDIA's nvcc compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. For this reason, we decided to write a compiler that transforms P2G programs into C++ files, which are further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control over the resulting object code, but we gain the flexibility and sophisticated optimization of the native platform compilers, and a much more lightweight P2G compiler. Depending on the target architecture, using the native compiler may be necessary to even run programs at all, since low-level documentation is not available. The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

Runtime The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors, using the implicit dependency graph formed by kernel definitions. However, at the time of writing, the prototype runtime does not yet have a full implementation of deadline expressions. This is because the semantics of the kernel language support for this feature are not fully defined yet.

The prototype targets a node with multiple processors. It is designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which may require resize operations. A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernel's *fetch* statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement, and puts these into a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

The low-level scheduler consists of a dependency analyzer and kernel instance dispatcher. Using the implicit dependency graph, the dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker threads. Dependencies are analyzed in a dedicated thread, which handles events emitted from running kernel instances whenerver *store* and *resize* operations are performed on fields. Kernel instances are executed by a worker thread dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements or by groups of kernels that satisfy their own dependencies in aging cycles, such as the *mul2* and *plus5* kernel in figure 3.6.

The runtime is written in C++ and uses the blitz++ [87] library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://www.p2gproject.org/.

3.5.2 Workloads

We have implemented a few workloads commonly used in multimedia processing to test the prototype implementation. The P2G kernel language is able to expose both the data and task parallelism of the programs to the P2G system, so the runtime is able to adapt execution of the programs to suit the target architecture.

K-means clustering K-means clustering is an iterative algorithm for cluster analysis that aims to partition n datapoints into k clusters where each datapoint belongs to the cluster with the nearest mean value. As shown in figure 3.8, the P2G k-means implementation consists of an

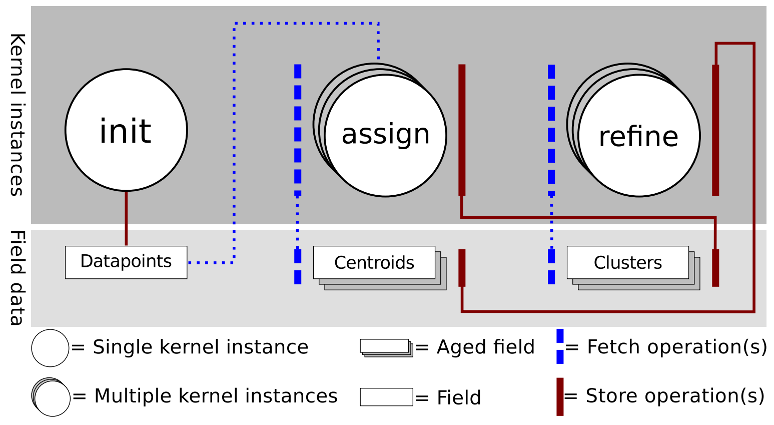


Figure 3.8: Overview of the K-means clustering algorithm

init kernel, which generates n datapoints and *stores* them in the datapoints field. Then, it selects k of these datapoints randomly, as the initial means, and *stores* them in the centroids field. Next, the *assign* kernel *fetches* a slice of data, a single datapoint per *kernel instance*, the last calculated centroids, and *stores* this datapoint in the cluster of the closest centroids using the Euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* form a loop that gradually leads to a convergence in centroids, at which point the k-means algorithm has completed.

Motion JPEG Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images. The MJPEG format provides many layers of parallelism, well-suited for illustrating the potential of the framework. We focused on optimizing the discrete cosine transform (DCT) and quantization part as this is the most compute-intensive part of the codec.

The *read* + *splitYUV* kernel reads the input video in YUV-format and stores the data in three global fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing into the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 3.9, we see that the respective DCT kernels depend on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma

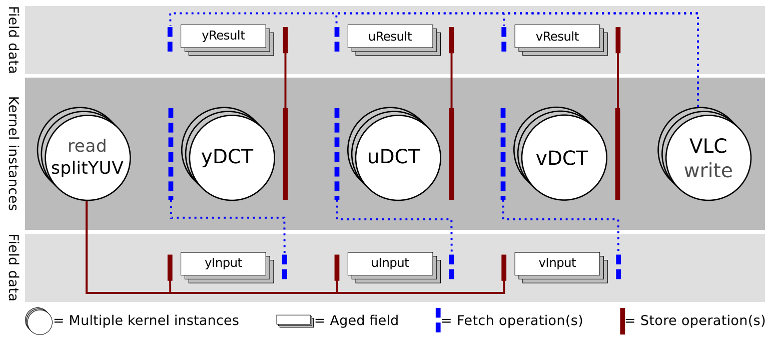


Figure 3.9: Overview of the MJPEG encoding process

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table 3.1: Overview of test machines

sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*, respectively. Finally, the *VLC* + *write* kernel stores the MJPEG bit-stream to disk.

3.5.3 Evaluation

We have run tests with the workloads MJPEG and K-means (described in section 3.5.2). Each test was run on a *4-way Core i7* and an *8-way Opteron* (see table 3.1 for hardware specifications) ranging from *1* worker thread to *8* worker threads with *10* repetitions per worker thread count. The results of these tests are reported in figures 3.11 and 3.10, which show the mean running time in *seconds* for each machine for a given thread count with standard deviation reported as error-bars.

In addition, we have performed micro-benchmarks for each workload, summarized in tables 3.2 and 3.3. The benchmarks summarize the number of kernel instances dispatched per kernel definition, dispatch overhead and time spent in kernel code.

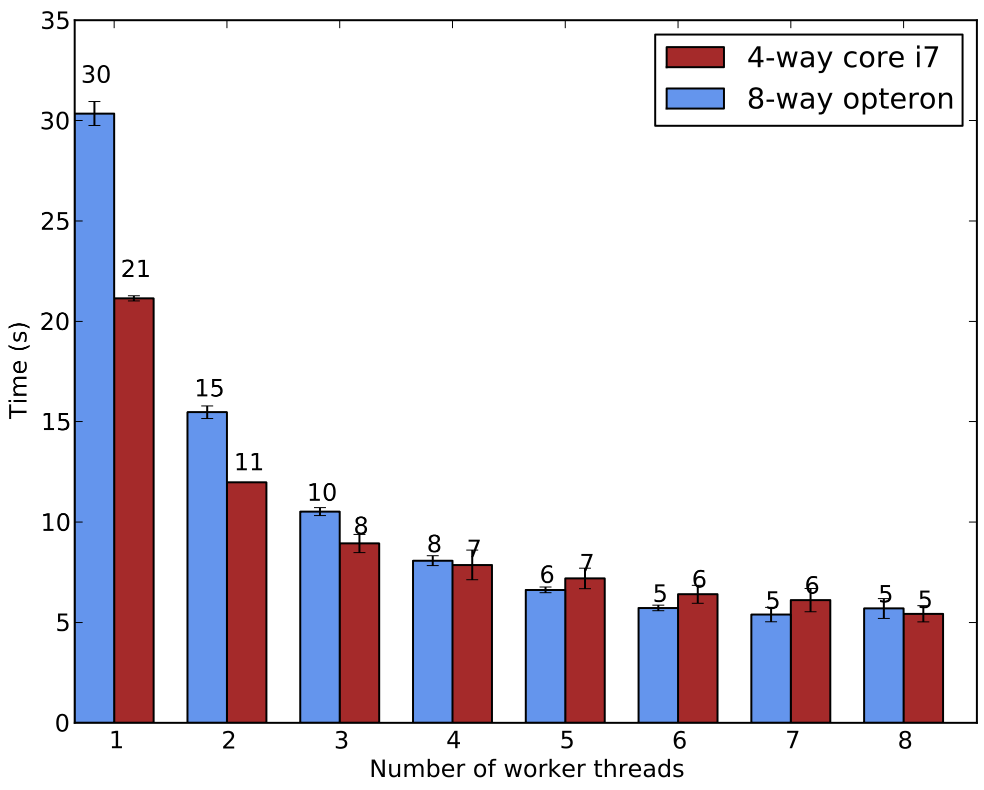


Figure 3.10: Workload execution time for Motion JPEG

Motion JPEG The Motion JPEG workload is run on the standard test sequence *Foreman* encoded in *CIF* resolution. We limited the workload to process *50* frames of video.

As we can observe in figure 3.10, P2G is able to scale close to linearly with the resources it has available. In P2G, the dependency analyzer of the low-level scheduler runs in a dedicated thread. This affects the running time when moving from *7* to *8* worker threads, where the eighth thread shares resources with the dependency analyzer. To compare, the standalone single threaded MJPEG encoder, upon which the P2G version is based, has a running time of *30* seconds on the Opteron machine and *19* seconds on the Core i7 machine. Note that both the standalone and P2G versions of the MJPEG encoder use a naive DCT calculation, there are versions of DCT that can improve performance significantly, such as FastDCT [29].

From table 3.2, we can see that time spent in kernel code is considerably higher than the dispatch overhead for the kernel definitions. The dispatch time includes allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have a considerably higher dispatch time than the **DCT* operations.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table 3.2: Micro-benchmark of MJPEG encoding in P2G

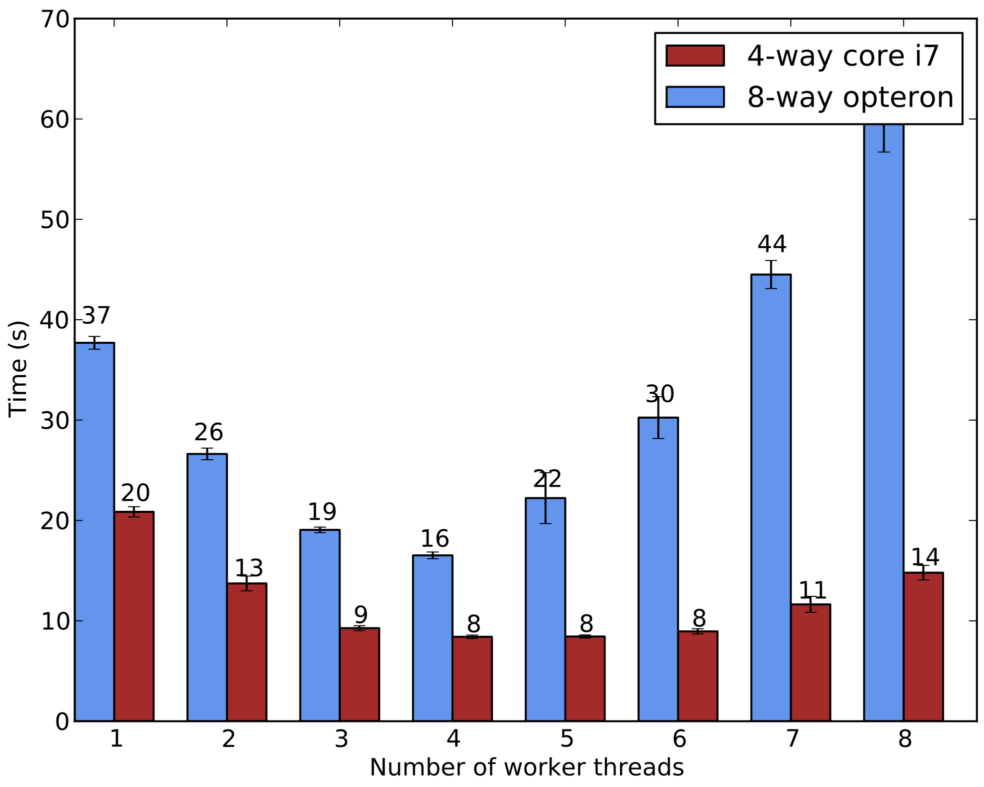


Figure 3.11: Workload execution time for *K*-means

We can also see that the most CPU-time is spent in the kernel instances of *yDCT*, *uDCT* and *vDCT*, which is the computationally intensive part of the workload. This indicates that decreasing data and task granularity, as discussed in section 3.4, has little impact on the throughput of the system. This is because most time is already spent in kernel code.

Note that even though there are *51* instances of the *read/write* kernel definitions, only *50* frames are encoded, because the last instance reaches the end of the video stream.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table 3.3: Micro-benchmark of k-means in P2G

K-means The K-means workload is run with K=*100* using a randomly generated data set containing *2000* datapoints. The K-means algorithm is not run until convergence, but with *10* iterations. If we do not define this break-point it is undefined when the algorithm converges, and we introduced this condition to ensure that we get a relatively stable running time for each run.

As seen in figure 3.11, the K-means workload scales to *4* worker threads. After this, the running time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition, as witnessed when comparing the dispatch time to the time spent in kernel code. This leads to the serial dependency analyzer becoming a bottleneck in the system. As discussed in section 3.4, this condition could be alleviated by decreasing the granularity of data-parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time, and reduce the workload of the dependency analyzer. The reduction in work for the dependency analyzer is a result of the lower number of kernel instances being run.

The two different test machines behave somewhat differently in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks, and we think this is why the Core i7 suffers less when we meet the limitations dictated by Amdahl's law. The considerable time *init* spends in kernel code is because it generates the data set.

3.5.4 Discussion

We have provided an implementation of a P2G execution node that together with our workloads demonstrates a real working system. The limitations of the prototype are obvious: no support for heterogeneous architectures, static scheduling, no distributed or deadline support and a dependency analyzer with limited scalability. Still, given the huge amount of time to evaluate design and implementation alternatives and the immense engineering effort required for implementing an unproven framework like P2G, we are satisfied with getting this far. We have succeeded at implementing a working prototype capable of executing multimedia workloads, and showed that it is possible to express workloads within the basic concepts of P2G. The runtime was a complex endeavor with several large components working together, including a dependency analyzer, compiler, code generator, field storage containers and more. The prototype implementation is a first attempt at leveraging the P2G concepts in a working system, and there is much ongoing work in further improving this system.

Usability One fundamental idea presented in this chapter is designed to expose all available parallelization opportunities of workloads to the P2G runtime using kernels and fields. Since there can be fewer computational resources available than parallelization opportunities, the framework is designed in such a way that the granularity is reduced to the parallel resources. This is in contrast to approaches such as auto parallelization, where parallel opportunities are inferred by dependencies, or conventional threading, where the parallel usage is static. The kernels and fields provide a way to expose parallelization opportunities, and to connect them to code and data. The kernel language is not a significant contribution in this work as we are not language designers and have little prior experience in this area. However, the kernel language allows workload writers to use the P2G framework with little knowledge of the internal details, and by enabling support for embedded code, provide an option for directly porting native C/C++ code to the P2G framework. We think this outweighs the burden of using a new language for workload writers, but have not investigated this further.

The main advantage of using the P2G framework instead of writing natively for the target architecture is portability. Everything that the framework provides can also be done by manually writing native code, for example with OpenMP, and even distributed support, e.g., with MPI. The portability allows a workload writer to expose her workload without targeting a specific architecture (or even microarchitecture as we saw), and let the framework adapt it to the machine. This is similar to a traditional compiler, but the P2G idea is to abstract the concepts further and let the runtime optimize the execution. Like Oracle's Java [88], we enable portability, but in a much more abstract manner, allowing executing on architectures with exotic memory layouts and instruction sets (heterogeneous architectures).

Proof of concept Many of the ideas of P2G are preliminary and untested, and especially the runtime is merely a proof of concept. We wanted to provide a framework for early and convenient experimentation of novel ideas for execution, scheduling, instrumentation and so forth, which will be future work built on top of the contributions in this thesis. As such, the prototype implementation presented is lacking in terms of P2G functionality and is not yet optimized for performance. Still, it provides a fundament for further experimentation and development of the core concepts. By developing the prototype, we tried many ways for designing a framework, resulting in a number of dead ends before we settled onto the design presented here.

One obvious question is why we did not base our work on an existing codebase or framework, both in terms of reducing engineering work and increasing usability. The answer is that we did not find anything that at the time could solve the challenges stated in the problem statement. Furthermore, we wanted to evaluate the concepts of kernel and fields to find out if we were able to express multimedia workloads within those concepts, and if they were usable as fundamental concepts. Another reason is that by having designed and implemented the entire system from scratch, we understand the issues better, enabling us isolate and understand results better. This will make it easier to contribute individual concepts, techniques and results from the P2G framework to other similar execution systems. For a production environment, designing and writing the framework from scratch is a modest idea, at best, but from our exploratory and experimentation approach, we were able to understand the system in a more complete manner by doing this.

Limitations of the framework The P2G framework language and runtime are designed for a particular set of workloads, i.e., continuous multimedia workloads with real time requirements running on heterogeneous architectures. This does not imply dense data processing, although the current prototype runtime supports only this, since the fields are implemented as matrices. The concept of fields does not limit the implementation, and a field could be implemented as a graph or even as a distributed hash table across several nodes. Experimenting with abstract concepts for fields is left as further work. The framework prototype currently uses a dependency analyzer running in a single thread and consumes a relatively large amount of computational resources. This can be improved by optimizing the dependency analyzer itself, but also by implementing granularity reduction such that kernel instances are grouped in larger units, limiting the number of dependencies.

The realtime aspect of the framework is also unfinished, with the timer interface not fully defined, and deadline avoidance and mitigation schemes to be developed. Still, the entire framework was designed with this in mind, and the initial ideas of P2G are continuously expanded as development continues.

Other workloads The P2G framework is designed for a particular set of workloads, i.e., cyclic multimedia data processing. This will typically mean transformation of dense data by applying a series of filters. An example of such a workload is SIFT [89] to extract image features, comprising a series of compute intensive filters in sequence. Although the P2G design can be used for such workloads, it may serve limited usefulness in describing other workloads. We have not thoroughly evaluated the implications of our design on many workloads, but have found that it can be used constructively for some workloads. Further evaluations of this are left as further work.

Feedback based scheduling We have not evaluated different scheduling strategies in the P2G framework yet, but we know the scheduler must have little overhead and perform well under different conditions. The fundamental concept for low level scheduling in P2G is reactive behaviour; by constantly instrumenting the system's behaviour, we plan to tune the framework's parameters based on feedback about the current state. The reason for this is observations demonstrated in chapter 2, wich show that the best order of execution, and correct granularity of data and code is next to impossible to predict in advance, and depends heavily on externalities such as system load and usage of shared resources, as well as the microarchitecture of choice. The details of such a scheduler are currently under investigation by another PhD candidate. We have earlier compared static scheduling algorithms such as graph/hypergraph partitioning, and compared this to work stealing-based scheduling in [90]. Graph partitioning and similar approaches require significant resources to resolve. They may be applicable for high level scheduling, which is not expected to require rescheduling very often. Scheduling on the node itself should benefit from a simpler heuristic-based scheduler that can take advantage of the runtime feedback to reduce the time between scheduling events as well as minimizing the computational cost of scheduling.

Run-time optimization The current prototype does not take advantage of the many opportunities for runtime optimizations. We expect programmers to write their programs in an architecture-independent manner in addition to exposing as much of the parallelization opportunities as possible. This means that efficient runtime optimization is imperative to achieve performance similar to hand optimized programs natively written for the target architecture. We saw this in the experiments, where the dispatch overhead for launching k-means kernel instances, together with the dependency analyzer inhibited the scalability of the system. In the prototype, kernel instances are created 1:1 with the expressions in the kernel language, also all fetches/stores incur actual memory transfers. The runtime implementation has significant freedom to transform these statements into optimized machine code, e.g., by merging kernel instances or by removing unnecessary memory transfers by analyzing what kernels that refers to what parts of the virtual fields. One of the most promising opportunities going forward in this regard is using just-in-time compilation (JIT) to compile the workloads dynamically as needed. Another PhD candidate and his students are currently investigating using LLVM [91] to add JIT compilation to the framework. Since this is ongoing work, we do not know how this will work, but we hope to get close to native performance combined with the advantages of elasticity and architecture abstractions.

Other optimization opportunities include the execution order of kernel instances as we saw in section 2.1.2. As it turned out, the better-performing order of execution did not only depend on the type of workload being executed, but also on the target microarchitecture and generation of processor at hand. We claim that predicting what performs better in advance is nearly impossible, and as such, is better left to a runtime scheduler that takes advantage of continuously generated instrumentation data. Investigating this in the P2G system is a very interesting area for going further, as the implications affect other systems as well.

3.5.5 Summary

With P2G, we have proposed a new flexible framework for automatic parallel, real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of data and task decomposition. Using our kernel language, this decomposition is expressed through kernel definitions and fetch and store statements on fields. This language is independent from the P2G runtime and can easily be replaced. A workload defined in our kernel language is compiled for execution in P2G. This workload can then be partitioned by the high-level scheduler of a P2G master node, which then distributes partitions to P2G execution nodes and runs the tasks locally. Execution nodes can consist of heterogeneous resources. A low-level scheduler at the execution nodes then adapts the partial (or full) workload to run optimally using resources at hand. Feedback from the instrumentation daemon at the execution node can lead to repartitioning of the workload (a task performed by the high-level scheduler). The aim is to bring the ease of batch-processing frameworks to multimedia workloads.

In this work, we have presented an execution node capable of running on a multi-way architecture. Our experiments running on a prototype show the potential of our ideas. However, there still remains a number of vectors for optimization. In the low-level scheduler, we have identified that combining task and data to minimize overhead introduced by P2G is a first reasonable modification. Additionally, completing the implementation of a fully distributed version is in the pipeline. Also, writing workloads for heterogeneous processing cores like GPUs and non-cache coherent architectures like IBM's Cell Architecture is a further consideration. Currently, we are investigating appropriate mechanisms for both high- and low-level scheduling, garbage collection, fat binaries, resource profiling and monitoring, and efficient migration of tasks.

While a number of optimizations remain, we have shown that P2G is feasible, through the implementation of this execution node, and the successful implementation of multimedia workloads, such as MJPEG and k-means. With these workloads, we have shown that it is possible to express multimedia workloads in the kernel language and we have implemented a prototype of an execution node in the P2G framework that is able to execute kernels and scales with the available resources.

Chapter 4

Papers and Contributions

4.1 Overview of the research papers

The research work done during my PhD period has revolved around systems support for multimedia, and has included a wide area of research, ranging from low level I/O scheduling and up to video codec optimizations. Although somewhat unstructured, it has enabled me to collaborate with several colleagues on highly interesting topics as we seized the opportunities when we saw them. Eventually, we focused on the topic of processing multimedia workloads on modern architectures, and have thus selected four papers as the main contributions to this thesis. Our nexus is the P2G framework [92], a system and language designed and built from scratch, allowing multimedia workloads to run on modern hardware in a portable manner and demonstrated live at Eurosys [50] and ACM Multimedia [93]. To support this, we also present three papers [16,28,42] about multimedia processing and architecture considerations on varying modern architectures for building a portable runtime. The findings in these latter papers allowed us to design the P2G system with enough flexibility and expressive power to support the workloads, as well as providing insights into how the runtime must be designed to run on different architectures. Although the other papers [90, 94--100] we wrote are multimedia systems related, we have limited the thesis to four papers as we find these easier to combine as a single work. The main papers are presented chronologically in the following sections.

4.2 Paper I: Transparent Protocol Translation for Streaming

Abstract The transport of streaming media data over TCP is hindered by TCP's probing behavior that results in the rapid reduction and slow recovery of the packet rates. On the other side, UDP has been criticized for being unfair against TCP connections, and it is therefore often blocked out in the access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently performs transport protocol translation. We translate HTTP requests by the client transparently into RTSP requests, and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side that scales to a higher load when it doesn't have to deal with TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side and data streams are passed through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP-streaming where the TCP bandwidth oscillates heavily.

Lessons learned This paper, written as a network research paper, explores using an asymmetric multicore processor (Intel IXP) to process a novel network protocol translation technique in real-time. Although, the main contribution of this paper is the network aspect, using the IXP architecture for real-time processing allowed us to do our technique on the data plane, hence providing minimal latency overhead and high bandwidth, compared to a software proxy. Experimenting with the IXP also gave us insights into the level of flexibility required for our P2G framework to properly function with such architectures. With IXP having multiple (incompatible) instruction sets and compilers, in combination with partly shared, partly exclusive memory mapped to the cores, having abstracted memory in our framework became obvious.

Author's Contributions Espeland contributed significantly to the overall design, implementation and evaluation of this work. Together with Lunde, he implemented the proxy from scratch. Further, Espeland designed the experimental setup and performed the experiments to evaluate the proposed solution. Evaluation of the experiments and the rest of the paper writing were done in collaboration with the other authors.

Published in Proceedings of the 15th International Multimedia Conference (MM), ACM, 2007.

4.3 Paper II: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Abstract When used efficiently, modern multicore architectures, such as Cell and GPUs, provide the processing power required by resource demanding multimedia workloads. However, the diversity of resources exposed to the programmers, intrinsically requires specific mindsets for efficiently utilizing these resources - not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion-JPEG implementations indicates that there exists a large potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices, inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks and troubles with respect to efficient utilization of the available resources.

Lessons learned The second paper is an architecture paper where we basically wanted to investigate if porting multimedia algorithms to different heterogeneous architectures would give acceptable performance and thus could be done in a suitable manner using cross compiling, just in time, or other techniques for efficient and portable execution of multimedia workloads on such architectures. Among the findings, we saw that although the theoretically most efficient algorithm performed best on one architecture, a very different (and theoretically slower) algorithm significantly outperformed this on other architectures. Thus, we could conclude that for running optimized and portable workloads in the P2G framework, we would need support for multiple versions of workloads providing different algorithmic implementations in our system. Although not yet realized in the prototype runtime, this was an intrinsic requirement in the framework. Re-usability of code became important at this point since one may have to integrate an existing code base into the runtime. Obviously, reusing existing code inhibits portability since it is not written in such a way that it can be optimized or ported by a runtime. Still, we saw this as an important feature and designed the framework to target existing compilers, allowing libraries to be linked in at the cost of portability. With such a design, critical parts can be exposed in our framework while parts that are not sensitive to performance can easily be reused in the system. Although inhibiting to portability, it makes the framework much more practical.

Author's Contributions Espeland contributed significantly to the design, implementation and evaluation of this work. Together with Stensland, he designed the experiments and evaluated the results. He performed much of the implementation work and wrote the paper text in collaboration with the other authors.

Published in The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), ACM, 2010.

4.4 Paper III: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Abstract The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason, Google and Microsoft implemented their respective processing frameworks MapReduce, as they allow the developer to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation in the design of these processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs, or even pre-determined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution. With the Nornir runtime system for parallel programs, which is a Kahn Process Network implementation, we addressed and solved several of these limitations. However, it is more difficult to use than other frameworks due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, P2G, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines, and provides both data- and task-parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We have implemented an (interchangeable) P2G to ease development. In this paper, we present a proof of concept implementation of a P2G execution node and some experimental examples using complex workloads like Motion JPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

Lessons learned The third paper presents the framework designed from all our ideas and experience collected through a wide area of multimedia systems research. Although the framework is far from complete, it provides an experimental platform for promising ideas and the prototype runtime evaluated as part of this work allowed us to try out our assumptions on a real system. One of the interesting things we discovered with the prototype was the effect of data set granularity and the execution order of kernel instances. Predicting what was the *correct* work size to maximize performance was hard, if not impossible, since it depends on so many issues, e.g., occupancy, cache coalescing, memory bandwidth and other shared resources in the system. Using feedback based scheduling, monitoring resources and making decisions on these, seemed to be the most promising way going forward. The predominant method of similar frameworks is work stealing, an algorithm providing little or no structural requirements to the order of execution of workloads. As such, we investigated this effect further in the next paper by isolating the scheduler from the framework. The P2G framework was also demonstrated live at Eurosys [50] and ACM Multimedia [93].

Author's Contributions Espeland contributed significantly to the design, implementation and evaluation of this work. Espeland and Beskow developed most of the P2G ideas together, while Beskow focused on the high-level concepts for distributed computing in P2G, Espeland designed and developed the low level mechanisms of the framework. Evaluation and microbenchmarking were done by Espeland and Beskow, and the paper text were written in collaboration with all of the authors.

Published in Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2011 International Conference on Parallel Processing Workshops, IEEE, 2011.

4.5 Paper IV: Low-level Scheduling Implications for Data-intensive Cyclic Workloads on Modern Microarchitectures

Abstract Processing data intensive multimedia workloads is challenging, and scheduling and resource management are vitally important for the best possible utilization of machine resources.

In earlier work, we have used work-stealing, which is frequently used today, and proposed improvements. We found already then that no singular work-stealing variant is ideally suited for all workloads. Therefore, we investigate in more detail in this paper how workloads consisting of various multimedia filter sequences should be scheduled on a variety of modern processor architectures to maximize performance. Our results show that a low-level scheduler additionally cannot achieve optimal performance without taking the specific micro-architecture, the placement of dependent tasks and cache sizes into account. These details are not generally available for application developers and they differ between deployments. Our proposal is therefore to use performance monitoring and dynamic adaption for the cyclic workloads of our target multimedia scenario, where operations are repeated cyclically on a stream of data.

Lessons learned In the fourth paper presented in this thesis, we looked at how execution order affects performance on modern x86 processors. We tested processors from two vendors using a set of multimedia workloads, and among the unexpected results, we found that the best execution order depends not only on the workload, but on the microarchitecture (vendor and generation specific), and the utilization of shared resources. Drawing conclusions on what is a single best scheduler in the P2G framework seemed futile, and this strengthen our belief that a feedback based approach, as suggested in paper III, where system resources are monitored is the way forward.

Author's Contributions Espeland contributed significantly to the design, implementation and evaluation of this work. Espeland designed and evaluated the experiments together with Olsen. Espeland wrote the evaluation framework while Olsen wrote most of the workloads. Writing the paper were done jointly by the authors.

Published in Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2012 International Conference on Parallel Processing Workshops (ICPPW), IEEE, 2012.

4.6 Other publications

Several other papers were published at journals and conferences in the PhD period. We did not include all of them here to limit the scope of this thesis. Instead, we will give a short summary of the contributions in them.

Nornir The first two papers, was included as part of Zeljko Vrba's thesis in published in 2009, introduced the Nornir framework [90] and exposed limitations to work stealing scheduling [94]. The papers serve as a motivation to the research presented in this thesis, and although I contributed to the papers, it was mainly Vrba's work and is as such not included in this thesis. The Nornir framework used Kahn Process Networks (KPN) for execution on multicore x86 machines and although they provided great flexibility, using KPNs was challinging for programmers writing workloads. Still, the flexibility of KPNs had a large impact on the design of our P2G framework. For instance, we found it imperative to have a practical kernel language, even for prototyping to handle more complex workloads. We also learned what type of schedulers are suitable in the P2G setting; graph partitioning schedulers provide great balance, but their computational overhead proved way too high for kernel instance scheduling. Also, the modifications to the work stealing scheduler inspired us to investigate the impact of execution order, resulting in paper IV and important knowledge for further designing a P2G scheduler.

Game server In another system called LEARS [96], we designed and evaluated a scheme for lockless scalability for gameservers. The approach allowed massively multiplayer online game servers with much more scalability than traditional gameservers and hence more players within an unpartitioned area. The LEARS prototype was demonstrated live at Netgames [100] in 2011. In the context of P2G, this type of scalable game server would be a perfect workload going further. MMO game servers require high levels of elasticity and high performance. Currently, the P2G prototype is not feature complete enough to support this workload, but should be so in the future.

Video Codec During our research with the VP8 codec in collaboration with group members doing research on adaptive streaming, we looked at the performance issues when encoding video segments at various quality layers [95, 97]. To address this, we proposed modifying the motion estimation algorithm of the encoder such that motion vectors are reused in different quality outputs. Although this leads to a slight degradation in quality, the performance benefit is huge with a significant reduction in processing demands. In this work, dynamic adaptation or elasticity is also key and experimenting with complex codecs in the P2G prototype is a clear goal going further.

Disk I/O Finally, we worked on I/O performance optimizations by improving file tree traversal performance by scheduling in user space [98, 99]. The proposed technique ordered directory tree requests by the logical block order on disk, significantly improving performance on file tree traversal operations. This optimization is not possible in kernel space using the standard POSIX API, since too few I/O operations are issued at a time for the scheduler to efficiently optimize tree traversal. With dirty file systems, we gained up to four times the performance compared to regular file tree traversal. In this work, we demonstrated clearly the advantage of having several schedulers, a low-level (OS) scheduler ordering disk I/O after block order within the I/O queue, but also (as we added) a high-level scheduler, sorting file requests with a complete overview of the big picture. This approach, translated to task scheduling, has been fully adapted to the P2G framework, separating a fast and efficient low-level scheduler with a more complex high-level scheduler with detailed system instrumentation.

Chapter 5

Conclusion

Working with modern architectures for high performance applications is increasingly more difficult for programmers as the complexity of both the system architectures and software continue to increase. As we showed in our Cell and GPU work [28], the level of hand tuning and native adaptations required to achieve performance comes at the cost of limiting the portability of the software. Several abstractions, languages and frameworks exist, but they are typically not written for cyclic multimedia workloads, and to shoehorn this type of workloads in frameworks designed for batch processing present a number of challenges by itself.

5.1 Summary

In this work, we have looked at considerations fundamental to achieving performance on prominent modern architectures and used this to design a framework for processing cyclic multimedia workloads. The results were presented in three case studies, analyzing architecture considerations for three machine architectures all requiring very different use of memory. In particular, we have used an MJPEG encoder as an example on Cell and GPU architectures to evaluate different ways to structure the workload on the architectures [28]. We also did experiments on an IXP2400 network processor with a novel application for transparently translating RTP/UDP to HTTP/TCP video streaming data in real time [41, 42]. Using the heterogeneous processing elements on the IXP allowed low latency and high bandwidth network traffic, but at considerable implementation complexity for such a specialized architecture resulting in very low portability of the workload to other platforms. In the last case study, we investigated how the execution order of order-independent elements affected performance and if we could observe differences between various microarchitectures of the same CPU family [16]. We found that, indeed, the best execution order was not static for a given workload and depended on the cache hierarchy that varied between CPU vendors and even CPU generations from the same vendor.

Using the knowledge gained from the case studies, we designed a framework for writing and executing multimedia workloads in an architecture-agnostic manner [50,92,93]. The framework distinguishes itself by exposing task, data and pipeline parallelism of the workloads in a way that allows the framework to tune granularity at compile or runtime instead of leaving this to the programmer. Further, the framework is specifically designed for multimedia workloads and provide intrinsic support for this domain with data abstractions and using time as a dependency. This allows the framework to exhibit great flexibility in execution of multimedia workloads to accommodate for heterogeneous architectures, fluctuating resource availability, real time support and even distributed execution. Some of the fundamental ideas of the framework were implemented in a prototype of the framework with a successful demonstration running on multicore x86. Although many aspects of the P2G ideas remains untested, it serves as a foundation for further experiments with executing pipelines of cyclic multimedia workloads on modern microarchitectures. The P2G project is still running, and currently, one PhD candidate and several students are working on refining the concepts and improving the prototype implementation, and it will be interesting to see what is possible with the framework.

5.2 Concluding remarks

Throughout this work, we have focused on system issues encountered when working with multimedia workloads on modern architectures. Since the commonly used and available architectures have diversified in the last years, frameworks, languages, middleware and other tools have and probably will become even more important for unifying the architecture differences. This is orthogonal to the recent popularity of batch processing frameworks such as MapReduce, Dryad, etc., which are designed for the age of *Big Data*, but do not solve the problems encountered with cyclic multimedia processing on modern architectures. As such, we believe this work addresses a rather unique problem.

With regard to performance, it is unrealistic to expect as good performance from a workload running in a framework, automatic optimized for a target architecture as a workload natively hand optimized by a competent programmer. It may be possible to provide directives that hint the system of what to do, but one problem with them is that since the language is architecture agnostic, they may not make sense in other configurations than what the programmer used. Further, in cases of resource sharing, virtualization and distributed execution, they may not improve performance at all since the conditions vary. A better approach may be to provide architecture specific optimizations that can be used by the runtime when applicable, even though they are not portable. We hope this should allow the framework to achieve close to the performance seen by natively written workloads.

Even though the P2G framework ideas and prototype are far from completed, we think we have made contributions to how such a framework should be designed and what considerations need to be applied for providing support for the diverse nature of modern microarchitectures. This is a field of rapid development and many active researchers and we expect many frameworks, languages and runtimes to come and go in this area. Although most of them will never be used in a production environment, they are essential for providing working knowledge on what is a good design and what is not, as defined in the *design paradigm* [14] of research methods.

Going back to the problem statement, we wanted to investigate if a pipeline of cyclic multimedia workloads could be executed by an elastic framework on modern microarchitectures in a portable and efficient manner. With this ambitions goal, we considered architectual details for doing so [16, 28, 41, 42], and in turn designed and implemented an execution framework for such workloads [50, 92, 93]. The framework is designed for portable and elastic execution, and although we have not implemented all features intended for the prototype yet, we have published an open source platform for evaluation and experimentation of the P2G ideas. Thus, the work presented here is proof-of-concept and shows that we can design and implement a framework that supports the target workload scenario.

5.3 Future work

The possibilities for future work in this domain are many, and we will only highlight a few that can be seen as likely next steps.

- From a low-level perspective, we see the scheduling of kernel instances as an interesting direction. From the work presented in chapter 2, we saw the importance the order of execution had, and that some execution orders even performed better on different microarchitectures. By using feedback-based scheduling using fine grained instrumentation data, we could design a scheduler that adapted to the available CPU resources at a very detailed level. Modern CPU instrumentation provide details of exact cache behaviour and memory transfers, which could be used as input to a smart low level scheduler.

- Another interesting venue is combing this with real time scheduling, using the provided deadlines in a speculative manner; high and low complexity versions of the workloads could be executed simultaneously (if resources permit), providing a fail safe if the high complexity version should not finish in time. With heterogeneous architectures, schedulers would have to take careful attention to what processors are used for what workloads and in particular what resources are exhausted benefits from clever co-scheduling.
- Extending or possibly rewriting our prototype for CUDA should also present interesting research opportunities, not to mention even more exotic architectures such as IBM's Cell architecture.
- Moving up in the stack, the framework is designed to support distributed execution, providing a whole new level of scheduling complexity. Scaling beyond a single machine is necessary to work with large data sets, particularly in real time, and designing two-level schedulers that efficiently work together in such a framework is likely to present a number of interesting research opportunities.

This is just a few of the ideas we have for future work within the P2G framework. We have spent considerable time developing the framework and prototype, and we look very much forward to see how the ideas presented here stand up to experiments going forward.

Bibliography

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

[2] Cisco. Cisco Visual Networking Index : Global Mobile Data Traffic Forecast Update , 2012 – 2017. Technical report, 2013.

[3] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: computing, communications, and applications*. Prentice-Hall International, 1995.

[4] Patrick Schmid. The Pentium D: Intel's Dual Core Silver Bullet Previewed, 2005.

[5] Nvidia. Nvidia cuda programming guide 2.3.1, August 2009.

[6] Intel. *Intel® IXP1200 Network Processor Family: Hardware Reference Manual*. Intel Corporation, 2001.

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

[8] G Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] Rajkumar Buyya, James Iroberg, and Andrzej M. Goscinski. *Cloud Computing: Principles and Paradigms*. John Whiley & Sons, 2011.

[10] Amazon Inc. *Amazon Elastic Compute Cloud (EC2)*. Amazon Inc., http://aws.amazon.com/ec2/, 2008.

[11] Apache Hadoop. `http://hadoop.apache.org/`.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10--10, 2004.

[13] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179--196, London, UK, UK, 2002. Springer-Verlag.

[14] D E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, and Paul R Young. Computing as a discipline. *Commun. ACM*, 32(1):9--23, January 1989.

[15] YouTube. Holy nyans! 60 hours per minute and 4 billion views a day on youtube. http://youtube-global.blogspot.com/2012/01/holy-nyans-60-hours-per-minute-and-4.html, January 2012.

[16] Havard Espeland, Preben N. Olsen, Pal Halvorsen, and Carsten Griwodz. Low-Level Scheduling Implications for Data-Intensive Cyclic Workloads on Modern Microarchitectures. In *2012 41st International Conference on Parallel Processing Workshops*, pages 332--339. Ieee, September 2012.

[17] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance Implications of Cache Affinity on Multicore Processors. *Lecture Notes in Computer Science*, 5168:151--161, 2008.

[18] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1--12, New York, NY, USA, 2000. ACM.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207--216, New York, NY, USA, 1995. ACM.

[20] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65--78, August 2009.

[21] Intel Corporation. Threading building blocks. `http://www.threadingbuildingblocks.org`.

[22] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 119--129, 1998.

[23] XIPH. *XIPH.org Test Media*, 2012. http://media.xiph.org/video/derf/.

[24] The Foundry. Nuke. http://www.thefoundry.co.uk/products/nuke/.

[25] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301--1321, 1991.

[26] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49--57, February 2010.

[27] IBM, Sony, and Toshiba. *Cell Broadband Engine Programming Handbook.* IBM, 2008.

[28] Håkon K. Stensland, Håvard Espeland, Carsten Griwodz, and Pål Halvorsen. Tips, tricks and troubles: optimizing for cell and gpu. In *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video*, pages 75--80. ACM Press, 2010.

[29] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of IEICE*, E71(11), 1988.

[30] M. Kovac and N. Ranganathan. JAGUAR: A fully pipelined VLSI architecture for JPEG image compression standard. *Proceedings of the IEEE*, 83(2), 1995.

[31] Ken Cabeen and Peter Gent. Image compression and the discrete cosine transform. In *Math 45*. College of the Redwoods.

[32] Westley Weimer, Michael Boyer, and Kevin Skadron. Automated dynamic analysis of cuda programs. In *Third Workshop on software Tools for MultiCore Systems (STMCS)*, 2008.

[33] Alexander Ottesen. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2009.

[34] Nvidia. Nvidia cuda c programming best practices guide 2.3, June 2009.

[35] Vipin Sachdeva, Michael Kistler, Evan Speight, and Tzy-Hwa Kathy Tzeng. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. *Parallel Computing*, 34(11):616--626.

[36] M.L. Curry, A. Skjellum, H.L. Ward, and R. Brightwell. Accelerating Reed-Solomon coding in RAID systems with GPUs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[37] Alexander van Amsesfoort, Ana Varbanescu, Henk J. Sips, and Rob van Nieuwpoort. Evaluating multi-core platforms for hpc data-intensive kernels. In *ACM Conference on Computing Frontiers (ICCF)*, 2009.

[38] Aleksandar Colic, Hari Kalva, and Borko Furht. Exploring NVIDIA-CUDA for video coding. In *ACM SIGMM conference on Multimedia systems (MMSys)*, 2010.

[39] Fabrizio Petrini, Gordon Fossuma, Juan Fernandez, Ana Lucia Varbanescu, Mike Kistler, and Michael Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, March 2007.

[40] Nvidia. Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. pages 1--16. Nvidia Whitepaper, 2011.

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

[42] Håvard Espeland, Carl Henrik Lunde, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Transparent protocol translation and load balancing on a network processor in a media streaming scenario. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, pages 129--130, New York, New York, USA, 2008. ACM Press.

[43] Buck Krasic and Jonathan Walpole. Priority-progress streaming for quality-adaptive multimedia. In *Proceedings of the ACM Multimedia Doctoral Symposium*, October 2001.

[44] Jörg Widmer, Robert Denda, and Martin Mauve. A survey on TCP-friendly congestion control. *Special Issue of the IEEE Network Magazine "Control of Best Effort Traffic"*, 15:28--37, February 2001.

[45] Peter Parnes, Kåre Synnes, and Dick Schefström. Lightweight application level multicast tunneling using mtunnel. *Computer Communication*, 21(515):1295--1301, 1998.

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

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

[48] Håvard Espeland, Paul B. Beskow, Hakon K. Stensland, Preben N. Olsen, Stale Kristoffersen, Carsten Griwodz, and Pal Halvorsen. P2G: A Framework for Distributed Real-Time Processing of Multimedia Data. In *2011 40th International Conference on Parallel Processing Workshops*, pages 416--426. Ieee, September 2011.

[49] Paul B. Beskow, Håkon K. Stensland, Håvard Espeland, Espen A. Kristiansen, Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodz, and Pål Halvorsen. Processing of multimedia data using the P2G framework. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, pages 819--820, New York, New York, USA, November 2011. ACM Press.

[50] Paul B Beskow, Håvard Espeland, Håkon K Stensland, Preben N Olsen, Ståle Kristoffersen, Espen A Kristiansen, Carsten Griwodz, and Pål Halvorsen. Distributed Real-Time Processing of Multimedia Data with the P2G Framework. Technical report, 2011.

[51] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59--72, New York, NY, USA, 2007. ACM.

[52] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123--1134, New York, NY, USA, 2008. ACM.

[53] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, pages 1099--1110, New York, NY, USA, 2008. ACM.

[54] C. Nicolaou. An architecture for real-time multimedia communication systems. *Selected Areas in Communications, IEEE Journal on*, 8(3):391--400, 1990.

[55] Ronnie Chaiken, Bob Jenkins, Per-Aake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265--1276, August 2008.

[56] Derek Murray, Malte Schwarzkopf, and Christopher Smowton. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[57] Daniel Waddington, Chen Tian, and KC Sivaramakrishnan. Scalable lightweight task management for mimd processors. In *Proceedings of Systems for Future Multi-Core Architectures (SFMA)*, 2011.

[58] Yuan Yu, Micahel Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Gunda, and Jon Currey. Dryadlinq: A system for general-puropose distributed data-parallel computing using a high-level language. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, 2008.

[59] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of IEEE HPCA*, pages 13--24, Washington, DC, USA, 2007. IEEE Computer Society.

[60] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[61] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of PACT*, pages 260--269, New York, NY, USA, 2008. ACM.

[62] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, pages 1029--1040, New York, NY, USA, 2007. ACM.

[63] Steffen Viken Valvåg and Dag Johansen. Oivos: Simple and efficient distributed data processing. In *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 113--122, 2008.

[64] Steffen Viken Valvåg and Dag Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *Proc. of IFIP International Conference on Network and Parallel Computing Workshops (NPC)*, pages 174--181, 2009.

[65] Ẑeljko Vrba, Pål Halvorsen, Carsten Griwodz, and Paul Beskow. Kahn process networks are a flexible alternative to mapreduce. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:154--162, 2009.

[66] Yingyi Blu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[67] Mark Thompson and Andy Pimentel. Towards multi-application workload modeling in sesame for system-level design space exploration. In Stamatis Vassiliadis, Mladen Berekovic, and Timo Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222--232. Springer Berlin / Heidelberg, 2007.

[68] Alex G. Olson and Brian L. Evans. Deadlock detection for distributed process networks. In *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, pages 73--76, 2006.

[69] E. A. De Kock, G. Essink, W. J. M. Smits, and P. Van Der Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000)*, pages 402--405. ACM Press, 2000.

[70] Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, Paul Beskow, Håvard Espeland, and Dag Johansen. The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce. *Journal of Sumpercomputing*, 27(1), 2010.

[71] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151--162, New York, NY, USA, 2006. ACM.

[72] T.M. Lee, E.A.; Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773--801, 1995.

[73] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[74] Joe Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1--6:26, 2007.

[75] Paul Hudakand John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1--12:55, 2007.

[76] ITU. *Z.100*, 2007. Specification and Description Language (SDL).

[77] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356--369, Washington, DC, USA, 2007. IEEE Computer Society.

[78] R.S.N. Arvind, R.S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598--632, 1989.

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

[80] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91--110, 2004.

[81] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[82] Guido Van Rossum and Fred L Drake. *The Python Library Reference*. Python Software Foundation, 2013.

[83] □Eljko Vrba, Pal Halvorsen, Carsten Griwodz, Paul Beskow, and Dag Johansen. The Nornir Run-time System for Parallel Programs Using Kahn Process Networks. *2009 Sixth IFIP International Conference on Network and Parallel Computing*, pages 1--8, October 2009.

[84] Lucas Roh, Walid A. Najjar, and A. P. Wim Böhm. Generation and quantitative evaluation of dataflow clusters. In *ACM FPCA: Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1993. ACM.

[85] Paul B. Beskow. *Parallel programming models and run-time system support for interactive multimedia applications*. PhD thesis, University of Oslo, March 2013.

[86] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, volume 5. Addison-Wesley Longman Publishing Co., Inc., 1995.

[87] Todd L. Veldhuizen. Arrays in blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 223--230, London, UK, 1998. Springer-Verlag.

[88] James Gosling, Bill Roy, Guy Steele, Gillad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 7 Edition*. Oracle America Inc., 2013.

[89] Park Jae-Han, Park Kyung-Wook, Baeg Seung-Ho, and Baeg Moon-Hong. SIFT: A Photometric and Scale Invariant Feature Transform. *International Conference on Pattern Recognition ICPR*, pages 1--4, 2008.

[90] Željko Vrba, Håvard Espeland, Pål Halvorsen, and Carsten Griwodz. Job Scheduling Strategies for Parallel Processing. 5798:280--299, October 2009.

[91] C Lattner and V Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization 2004 CGO 2004*, 57(c):75--86, 2004.

[92] Håvard Espeland, Paul B. Beskow, Hakon K. Stensland, Preben N. Olsen, Stale Kristoffersen, Carsten Griwodz, and Pal Halvorsen. P2G: A Framework for Distributed Real-Time Processing of Multimedia Data. In *2011 40th International Conference on Parallel Processing Workshops*, pages 416--426. Ieee, September 2011.

[93] Paul B. Beskow, Håkon K. Stensland, Håvard Espeland, Espen A. Kristiansen, Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodz, and Pål Halvorsen. Processing of multimedia data using the P2G framework. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, pages 819--820, New York, New York, USA, November 2011. ACM Press.

[94] Ž Vrba, H Espeland, P Halvorsen, and Carsten Griwodz. Limits of work-stealing scheduling. In *Job Scheduling Strategies …*, 2009.

[95] Håvard Espeland, Håkon Kvale Stensland, Dag Haavi Finstad, and Pål Halvorsen. Reducing Processing Demands for Multi-Rate Video Encoding. *International Journal of Multimedia Data Engineering and Management*, 3(2):1--19, 2012.

[96] Kjetil Raaen and Havard Espeland. LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition. In *Parallel Processing …*, pages 382--389. Ieee, September 2012.

[97] Dag H. Finstad, Hakon K. Stensland, Havard Espeland, and Pal Halvorsen. Improved Multi-Rate Video Encoding. In *2011 IEEE International Symposium on Multimedia*, pages 293--300. Ieee, December 2011.

[98] Carl Henrik Lunde, Håvard Espeland, Håkon Kvale Stensland, and Pal Halvorsen. Improving file tree traversal performance by scheduling I/O operations in user space. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 145--152. Ieee, December 2009.

[99] Carl Henrik Lunde, Håvard Espeland, Håkon K. Stensland, Andreas Petlund, and Pål Halvorsen. Improving disk i/o performance on linux. In *UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[100] Kjetil Raaen, Havard Espeland, Hakon Kvale Stensland, Andreas Petlund, Pal Halvorsen, and Carsten Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS). In *2011 10th Annual Workshop on Network and Systems Support for Games*, pages 1--3. Ieee, October 2011.

Part II

Research Papers

Paper I: Transparent Protocol Translation for Streaming

Title: Transparent Protocol Translation for Streaming [42].

Authors: H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 15th International Multimedia Conference (MM), ACM, 2007.

Transparent Protocol Translation for Streaming

Håvard Espeland[1], Carl Henrik Lunde[1], Håkon Kvale Stensland[1,2], Carsten Griwodz[1,2], Pål Halvorsen[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
{haavares, chlunde, haakonks, griff, paalh}@ifi.uio.no

ABSTRACT

The transport of streaming media data over TCP is hindered by TCP's probing behavior that results in the rapid reduction and slow recovery of the packet rates. On the other side, UDP has been criticized for being unfair against TCP connections, and it is therefore often blocked out in the access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently performs transport protocol translation. We translate HTTP requests by the client transparently into RTSP requests, and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side that scales to a higher load when it doesn't have to deal with TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side, and data streams are passed through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP-streaming where the TCP bandwidth oscillates heavily.

Categories and Subject Descriptors

D.4.4 [**OPERATING SYSTEMS**]: Communications Management—*Network communication*

General Terms

Measurement, Performance

1. INTRODUCTION

Streaming services are today almost everywhere available. Major newspapers and TV stations make on-demand and live audio/video (A/V) content available, video-on-demand services are becoming common and even personal media are frequently streamed using services like pod-casting or uploading to streaming sites such as YouTube.

The discussion about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability and jitter than could not be offered by TCP. Today, this approach is impeded with filters in Internet service providers (ISPs), by firewalls in access networks and on end-systems. ISPs reject UDP because it is not fair against TCP traffic, many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP-streaming, which delivers streaming media over TCP. The disadvantage is that the end-user can experience playback hiccups and quality reductions because of the probing behavior of TCP, leading to oscillating throughput and slow recovery of the packet rate. A sender that uses UDP would, in contrast to this, be able to maintain a desired constant sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that performs a transparent protocol translation. This is similar to the use of proxy caching that ISPs employ to reduce their bandwidth, and we do in fact aim at a combined solution. There are, however, too many different sources for adaptive streaming media that end-users can retrieve data from to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves a high perceived quality to end-users. Our prototype proxy is implemented on an Intel IXP2400 network processor and enables the server to use UDP at the server side and TCP at the client side.

We have earlier shown the benefits of combining the use of TFRC in the backbone with the use of TCP in access networks [1]. In the experiments presented in that paper, we used course-grained scalable video (scalable MPEG (SPEG) [4]) which makes it possible to adapt to variations in the packet rate. To follow up on this idea, we describe in this paper our IXP2400 implementation of a dynamic transport protocol translator. Preliminary tests comparing HTTP video streaming from a web-server and RTSP/RTP-streaming from the komssys video server show that, in case of some loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at play out rate while the TCP stream oscillates heavily.

2. RELATED WORK

Proxy servers have been used for improved delivery of

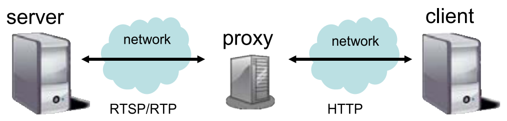


Figure 1: System overview

streaming media in numerous earlier works. Their tasks include caching, multicast, filtering, transcoding, traffic shaping and prioritizing. In this paper, we want to draw attention to issues that occur when a proxy is used to translate transport protocols in such a way that TCP-friendly transports mechanisms can be used in backbone networks and TCP can be used in access networks to deliver streaming video through firewalls. Krasic et al. argue that the most natural choice for TCP-friendly traffic is using TCP itself [3]. While we agree in principle, their priority progress streaming approach requires a large amount of buffering to hide TCP throughput variations. In particular, this smoothing buffer is required to hide the rate-halving and recovery time in TCP's normal approach of probing for bandwidth which grows proportionally with the round-trip time. To avoid this large buffering requirement at the proxy, we would prefer an approach that maintains a more stable packet rate at the original sender. The survey of [7] shows that TFRC is a reasonably good representative of the TCP-friendly mechanisms for unicast communication. Therefore, we have chosen this mechanism for the following investigation.

With respect to the protocol translation that we describe here, we do not know of much existing work, but the idea is similar to the multicast-to-unicast translation [6]. We have also seen voice-over-IP proxies translating between UDP and TCP. In these examples, a packet is translated from one type to another to match the various parts of the system, and we here look at how such an operation performs in the media streaming scenario.

3. TRANSLATING PROXY

An overview of our protocol translating proxy is shown in figure 1. The client and server communicates by the proxy, which transparently translates between HTTP and RTSP/RTP. Both peers are unaware of each other.

The steps and phases of a streaming session follows. The client tries to set up a HTTP streaming session, by initiating a TCP connection to the server. All packets are intercepted by the proxy, and modified before passing it on to the streaming server. The proxy also forwards the TCP 3-way handshake between client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connections. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file using RTP/UDP. The UDP packets are translated by the proxy as part of the

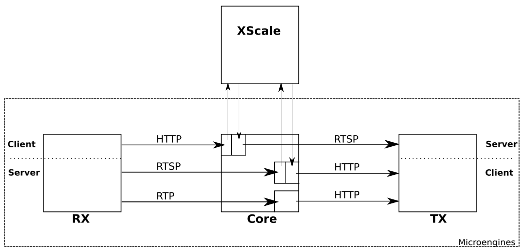


Figure 2: Packet flow on the IXP2400

HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

4. IMPLEMENTATION

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [5], which is designed to handle a wide range of access, edge and core applications. The basic features include a 600 MHz XScale core running Linux, eight 600 MHz special packet processors called micro-engines (μEngines), several types of memory and different controllers and busses. With respect to the different CPUs, the XScale is typically used for the control plane (slow path) while μEngines perform general packet processing in the data plane (fast path).

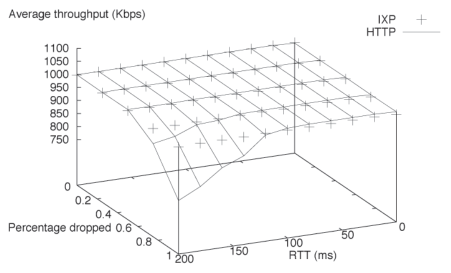
The transport protocol translation operation[1] is shown in figure 2. The protocol translation proxy uses the XScale core and one μEngine application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. RTSP and HTTP packets are enqueued for processing on the XScale core (control path) while the handling of RTP packets is performed on the μEngine (fast path). TCP acknowledgements with zero payload size are processed on the μEngine for performance reasons.

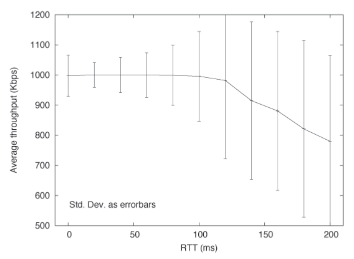
The main task of the XScale is to set up and maintain streaming sessions, but after the initialization, all video data is processed (translated and forwarded) by the μEngine. The proxy supports a partial TCP/IP implementation, covering only basic features. This is done to save both time and resources on the proxy.

To be fair with competing TCP streams, we implemented congestion control for the client loss experiment. TFRC [2] computation is used to determine the bandwidth available for streaming from the server. TFRC is a specification for best effort flows competing for bandwidth, designed to be reasonable fair to other TCP flows. The outgoing bandwidth is limited by the following formula:

$$X = \frac{s}{R * \sqrt{2 * b * \frac{p}{3}} + (t_{RTO} * 3 * \sqrt{3 * b * \frac{p}{8}} * p * (1 + 32 * p^2))}$$

[1]Our proxy also performs proxying of normal RTSP sessions and transparent load balancing between streaming servers, but this is outside of the scope of this paper. We also have unused resources (μEngines) enabling more functionality.


(a) HTTP and translation results


(b) HTTP streaming

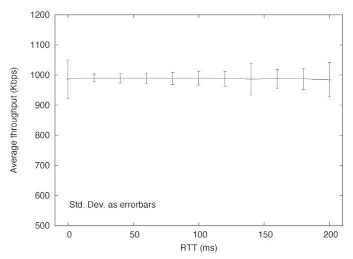

(c) Protocol translation

Figure 3: Achieved bandwidth varying drop rate and link latency with 1% server-proxy loss

where X is the transmit rate in bytes per second, s is the packet size in bytes, R is the RTT in seconds, b is the number of packets ACKed by a single TCP acknowledgment, p is the loss event rate (0-1.0), and t_{RTO} is the TCP retransmission timeout. The formula is calculated on a μEngine using fixed point arithmetic. Packets arriving at a rate exceeding the TFRC calculated threshold are dropped.

We are aware that this kind of dropping has different effects on the user-perceived quality than sender-side adaptation. We have only made preliminary investigations on the matter and leave it for future work. In that investigation, we will also consider the effect of buffering for at most 1 RTT.

5. EXPERIMENTS AND RESULTS

We investigated the performance of our protocol translation proxy compared to plain HTTP-streaming in two different settings. In the first experiment, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected proxy and client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0 - 200 ms) and the packet drop rate (0 - 1 %). We used a web-server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy whereas packets belonging to sessions initiated by connection made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump.

5.1 Server-Proxy Losses

The results from the test where we introduced loss and delay between server and proxy are shown in figure 3. Figure 3(a) shows a 3D plot where we look at the latency that we achieved for the different combinations ofl oss and link delays. Additionally, figures 3(b) and 3(c) show the respective results for the HTTP and protocol translation scenarios when keeping the loss rate constant at 1% (keeping the link delay constant gives similar results). The plots show that our proxy that translates transparently from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase. From figures 3(b) and 3(c), we see also that the translation provides a smoother stream whereas the bandwidth oscillates heavily using TCP end-to-end.

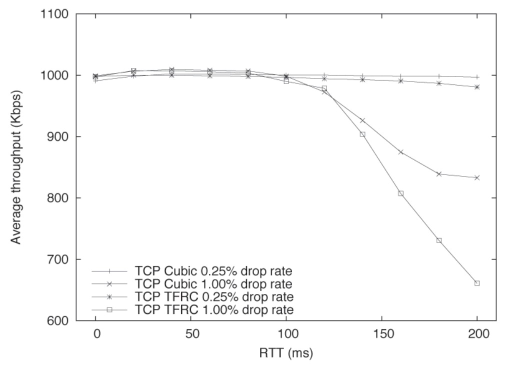


Figure 4: TCP cubic congestion vs. TFRC

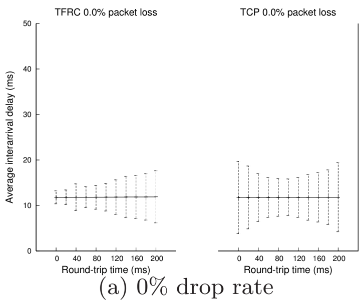
5.2 Proxy-Client Losses

In the second experiment, loss and delay are introduced between the proxy and the client, and the data rate is limited according to TFRC measuring RTT and packet loss during the transfer. Furthermore, packets are not buffered on the network card, meaning that the traffic exceeding the calculated rate of TFRC are dropped and that TCP retransmissions contains only data with zero values.

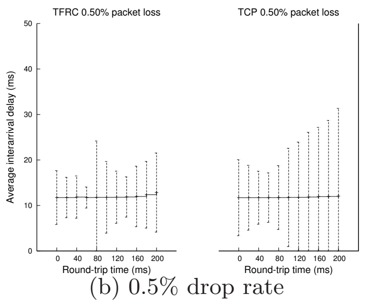
In figure 4, we first show the average throughput of streaming video from a web-server using cubic TCP congestion control compared with our TCP implementation using TFRC. As expected, the TFRC implementation behaves similar (fair) to normal TCP congestion control with a slightly more pessimistic approach. Moreover, figure 5 is a plot of the received packets' interarrival time. This shows that the delay variation of normal TCP congestion control increases with the drop rate, while TFRC is less affected. Thus, we see again that that our proxy gives a stream without large variations whereas the bandwidth oscillates heavily using TCP throughout the path.

6. DISCUSSION

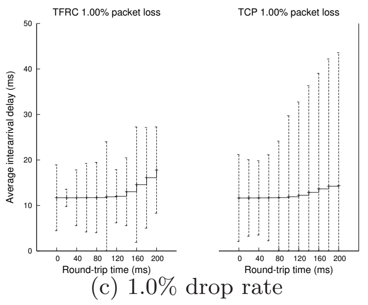
Even though our proxy seems to give better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packet (and thus old data if the client does not buffer), the proxy fills the new packet with new updated data from the server. This means that the client in our pro-



(a) 0% drop rate



(b) 0.5% drop rate



(c) 1.0% drop rate

Figure 5: Average interarrival delay and variation with proxy-client loss

totype does not receive all data, and some artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed play out. This can cause video artifacts, depending on the codec used. However, this problem can easily be reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how it should be implemented. For this study, we have chosen the IXP2400 platform as we earlier have explored the offloading capabilities of such programmable network processors. Using such an architecture, the network processor is suited for many similar operations, and the host computer could manage the caching and persistent storage of highly popular data served from the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding.

The main advantage of the scheme proposed in this paper is a lower variation in bandwidth and interarrival times in an unreliable network compared to normal TCP. It also combines some of the benefits of HTTP streaming (firewall traversal, client player support) with the performance of RTP streaming. The price of this is uncontrolled loss of data packets that may impact the perceived video quality more strongly than hiccups.

HTTP streaming may perform well in a scenario where a stored multimedia object is streamed to a high capacity end-system. Here, a large buffer may add a small, but acceptable, delay to conceal losses and oscillating resource availability. However, in the case where the receiver is a small device like a mobile phone or a PDA with a limited amount of resources, or in an interactive scenario like conferencing applications where there is no time to buffer, our protocol translation mechanisms could be very useful.

The server-proxy losses test can be related to a case where the camera on a mobile phone is used for streaming. Mobile devices are usually connected to unreliable networks with high RTT. The proxy-client losses test can be related to a traditional video conference scenario.

In the experiment, we compare a normal web-server streaming video with a RTP server (komssys) to a client by encapsulating the video data in HTTP packets on a IXP network card close to the video server. The former setup runs a simple web-server on Linux, limiting the average bandwidth from user-space to the video's bit rate.

Using RTP/UDP from the server through the backbone to a proxy is also an advantage for the resource utilization. RTP/UDP packets reduce memory usage, CPU usage and overhead in the network compared to TCP. This combined with the possibility of sending a single RTP/UDP stream to the proxy, and make the proxy do separation and adaptation of the stream to each client can reduce the load in the backbone. Therefore the proxy should be placed as close to the clients as possible, e.g. in the ISP's access network, or in a mobile provider's network.

7. CONCLUSION

Both TCP and UDP have their strengths and weaknesses. In this paper, we use a proxy that performs transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enables the server to use UDP on the server side and TCP on the client side. The server gains scalability by not having to deal with TCP processing. On the client side, the TCP stream is not discarded and passes through firewalls. The experimental results show that our protocol transparent proxy achieves translation and delivers smoother streaming than HTTP-streaming.

8. REFERENCES

[1] Griwodz, C., Fiksdal, S., and Halvorsen, P. Translating scalable video streams from wide-area to access networks. *Campus Wide Information Systems 21*, 5 (2004), 205–210.

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

[3] Krasic, B., and Walpole, J. Priority-progress streaming for quality-adaptive multimedia. In *Proceedings of the ACM Multimedia Doctoral Symposium* (Oct. 2001).

[4] Krasic, C., Walpole, J., and Feng, W.-C. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (2003), pp. 112–121.

[5] Intel Corporation. Intel IXP2400 network processor datasheet, Feb. 2004.

[6] Parnes, P., Synnes, K., and Schefström, D. Lightweight application level multicast tunneling using mtunnel. *Computer Communication 21*, 515 (1998), 1295–1301.

[7] Widmer, J., Denda, R., and Mauve, M. A survey on TCP-friendly congestion control. *Special Issue of the IEEE Network Magazine "Control of Best Effort Traffic" 15* (Feb. 2001), 28–37.

Paper II: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Title: Tips, Tricks and Troubles: Optimizing for Cell and GPU [28].

Authors: H. K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen.

Published: The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), ACM, 2010.

Tips, Tricks and Troubles: Optimizing for Cell and GPU

Håkon Kvale Stensland, Håvard Espeland, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{haakonks, haavares, griff, paalh}@simula.no

ABSTRACT

When used efficiently, modern multicore architectures, such as Cell and GPUs, provide the processing power required by resource demanding multimedia workloads. However, the diversity of resources exposed to the programmers, intrinsically requires specific mindsets for efficiently utilizing these resources - not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion-JPEG implementations indicates that there exists a large potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices, inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks and troubles with respect to efficient utilization of the available resources.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUE**]: Concurrent Programming—*Parallel programming*

General Terms

Measurement, Performance

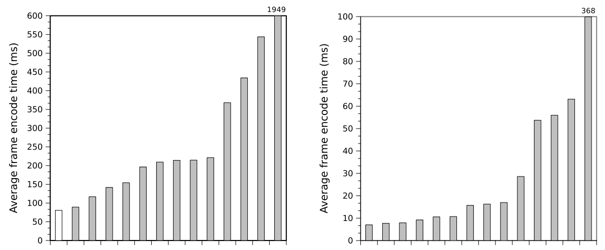
1. INTRODUCTION

Heterogeneous systems like the STI Cell Broadband Engine (Cell) and PCs with Nvidia graphical processing units (GPUs) have recently received a lot of attention. They provide more computing power than traditional single-core systems, but it is a challenge to use the available resources efficiently. Processing cores have different strengths and weaknesses than desktop processors, the use of several different types and sizes of memory is exposed to the developer, and limited architectual resources require considerations concerning data and code granularity.

We want to learn how to *think* when the multicore system at our disposal is a Cell or a GPU. We aim to understand how to use the resources efficiently, and point out tips, tricks and troubles, as a small step towards a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we have looked at effective programming for the workload-intensive yet relatively straight-forward Motion-JPEG (MJPEG) video encoding task. Here, a lot of CPU cycles are consumed in the sequential discrete cosine transformation (DCT), quantization and compression stages. On single core systems, it is almost impossible to process a 1080p high definition video in real-time, so it is reasonable to apply multicore computing in this scenario.

Our comparison of 14 different implementations on both Cell and GPU gives a good indication that the two considered architectures are complex to use, and that achieving high performance is not trivial. Derived from a sequential codebase, these multicore implementations differ in terms of algorithms used, resource utilization and coding efficiency. Figure 1 shows performance results for encoding the "tractor" video clip[1] in 4:2:0 HD. The differences between the fastest and slowest solution are 1869 ms and 362 ms per frame on Cell and GPU, respectively, and it is worth noting that the fastest solutions were disk I/O-bound. To gain experience of what works and what does not, we have examined these solutions. We have not considered coding style, but revisited algorithmic choices, inter-core data communication (memory transfers) and use of architecture-specific capabilities.

In general, we found that these architectures have large potentials, but also many possible pitfalls, both when choosing specific algorithms and for implementation-specific decisions. The way of thinking cross-platform is substantially different, making it an art to use them efficiently.



(a) Cell on PS3 (6 SPEs) (b) GPU on GTX 280

Figure 1: Runtime for MJPEG implementations.

[1]Available at ftp://ftp.ldv.e-technik.tu-muenchen.de/dist/test_sequences/1080p/tractor.yuv

2. BACKGROUND

2.1 SIMD and SIMT

Multimedia applications frequently perform identical operations on large data sets. This has been exploited by bringing the concept of SIMD (single instruction, multiple data) to desktop CPUs, as well as the Cell, where a SIMD instruction operates on a short vector of data, e.g., 128-bits for the Cell SPE. Although SIMD instructions have become mainstream with the earliest Pentium processors and the adoption of PowerPC for MacOS, it has remained an art to use them. On the Cell, SIMD instructions are used explicitly through the vector extensions to C/C++, which allow basic arithmetic operations on vector data types of intrinsic values. It means that the programmer can apply a sequential programming model, but needs to adapt memory layout and algorithms to the use of SIMD vectors and operations.

Nvidia uses an abstraction called SIMT (single-instruction, multiple thread). SIMT enables code that uses only well-known intrinsic types but that can be massively threaded. The runtime system of the GPU schedules these threads in groups (called warps) whose optimal size is hardware-specific. The control flow of such threads can diverge like in an arbitrary program, but this will essentially serialize all threads of the block. If it does not diverge and all threads in a group execute the same operation or no operation at all in a step, then this operation is performed as a vector operation containing the data of all threads in the block.

The functionality that is provided by SIMD and SIMT is very similar. In SIMD programming, vectors are used explicitly by the programmer, who may think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types.

2.2 STI Cell Broadband Engine

The Cell Broadband Engine is developed by Sony Computer Entertainment, Toshiba and IBM. As shown in Figure 2, the central components are a Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPE) connected by the Element Interconnect Bus (EIB). The PPE contains a general purpose 64-bit PowerPC RISC core, capable of executing two simultaneous hardware threads. The main purpose of the PPE is to control the SPEs, run an operating system and manage system resources. It also includes a standard Altivec-compatible SIMD unit. An SPE contains a Synergistic Processing Unit and a Memory Flow controller. It works on a small (256KB) very fast memory, known as the local storage, which is used both for code and data without any segmentation. The Memory Flow Controller is used to transfer data between the system memory and local storage using explicit DMA transfers, which can be issued both from the SPE and PPE.

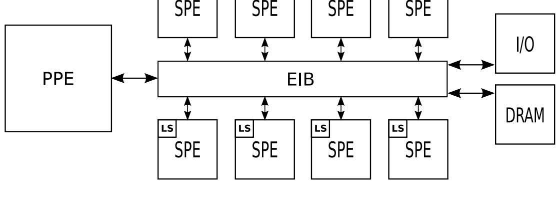


Figure 2: Cell Broadband Engine Architecture

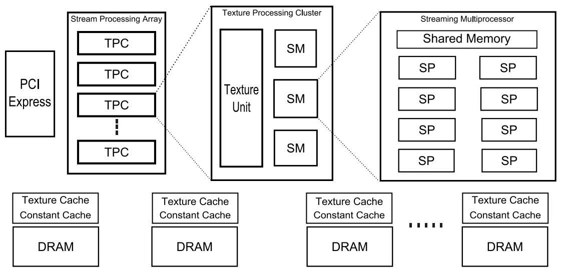


Figure 3: Nvidia GT200 Architecture

2.3 Nvidia Graphics Processing Units

A GPU is a dedicated graphics rendering device, and modern GPUs have a parallel structure, making them effective for doing general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia has released the CUDA framework with a programming language similar to ANSI C. In CUDA, the SIMT abstraction is used for handling thousands of threads.

The latest generation available from Nvidia (GT200) is shown in Figure 3. The GT200 chip is presented to the programmer as a highly parallel, multi-threaded, multi-core processor - connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters (TPC) with 3 streaming multiprocessors (SM). A single SM contains 8 stream processors (SP) which are the basic ALUs for doing calculations. GPUs have other memory hierarchies than an x86 processor. Several types of memory with different properties are available. An application (*kernel*) has exclusive control over the memory. Each thread has a private local memory, and the threads running on the same stream multiprocessor (SM) have access to a shared memory. Two additional read-only memory spaces called constant and texture are available to all threads. Finally, there is the global memory that can be accessed by all threads. Global memory is not cached, and it is important that the programmer ensures that running threads perform *coalesced* memory accesses. Such a coalesced memory access requires that the threads' accesses occur in a regular pattern and creates one large access from several small ones. Memory accesses that cannot be combined are called uncoalesced.

3. EXPERIMENTS

By learning from the design choices of the implementations in Figure 1, we designed experiments to investigate how performance improvements were achieved on both Cell and GPU. We wanted to quantify the impact of design decisions on these architectures.

All experiments encode HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* test sequence. However, we used only the first frame of the sequence and encode it 1000 times in each experiment to overcome the disk I/O bottleneck limit. This becomes apparent at the highest level of encoding performance since we did not have a high bandwidth video source available. All programs have been compiled with the highest level of compiler optimizations using gcc and nvcc, respectively, for Cell and GPU. The Cell experiments have been tested on a QS22 bladeserver (8 SPEs, the results from Figure 1 were on a PS3 with 6 SPEs) and the GPU experiments on a GeForce GTX 280 card.

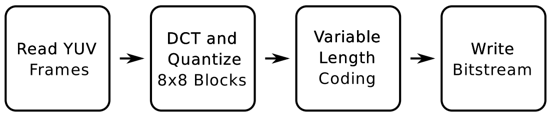


Figure 4: Overview of the MJPEG encoding process

3.1 Motion JPEG Encoding

The MJPEG format is widely used by webcams and other embedded systems. It is similar to videocodecs such as Apple ProRes and VC-3, used for video editing and postprocessing due to their flexibilty and speed, hence the lack of inter-prediction between frames. As shown in Figure 4, the encoding process of MJPEG comprises splitting the video frames in 8x8 macroblocks, each of which must be individually transformed to the frequency domain by forward discrete cosine transform (DCT) and quantized before the output is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC, our encoder uses predefined Huffman tables for compression of the DCT coefficients of each macroblock. The VLC step is not context adaptative, and macroblocks can thus be compressed independently. The length of the resulting bitstream, however, is probably not a multiple of eight, and most such blocks must be bit-shifted completely when the final bitstream is created.

The MJPEG format provides many layers of parallelism; Starting with the many independent operations of calulating DCT, the macroblocks can be transformed and quantized in arbitrary order, also frames and color components can be encoded separately. In addition, every frame is entropy-coded separately. Thus, many frames can be encoded in parallel before merging the resulting frame output bitstreams. This gives a very fine-level granularity of parallel tasks, providing great flexibility in how to implement the encoder. It is worth noting that many problems have much tighter data dependencies than we observe in the MJPEG case, but the general ideas for optimizing individual parts pointed out in this paper stand regardless of whether the problem is limited by dependencies or not.

The forward 2D DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D-plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label 1D-plain uses two consecutive 1D transformations with a transpose operation in between and after. This avoids symmetries, and the 1D transformation can be optimized further. One optimization uses the AAN algorithm, originally proposed by Arai et al. [1] and further refined by Kovac and Ranganathan [5]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the postscale operation, and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [2].

More algorithms for calculating DCT exist, but they are not covered here. We have implemented the different DCT algorithms as scalar single-threaded versions on x86 (Intel Core i5 750). The performance details for encoding HD video were captured using oprofile and can be seen in Figure 5. The plot shows that the 1D-AAN algorithm using two transpose operations was the fastest in this scenario, with the 2D-matrix version as number two. The average encoding time for a single frame using 2D-plain is more than 9 times that of a frame encoded using 1D-AAN. For all algorithms, the DCT step consumed most CPU cycles.

3.2 Cell Broadband Engine Experiments

Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts is available for mapping the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but also the entropy coding step can run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores, and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We solved this by adding a frame queue between the frame reader and the DCT step, and another queue between the DCT and VLC steps. Since a frame is processed in full by a single processor, the AAN algorithm is well suited for the Cell. It can be implemented in a straight-forward manner for running on SPEs, with VLC coding placed on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD) resulting in double encoding throughput, which can be seen in Figure 6 (Scalar- and Vector/PPE).

Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk in addition to dispatching jobs to SPEs. To be able to do this, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, i.e., two rows of luma and a single row of both chroma channels. The results show that

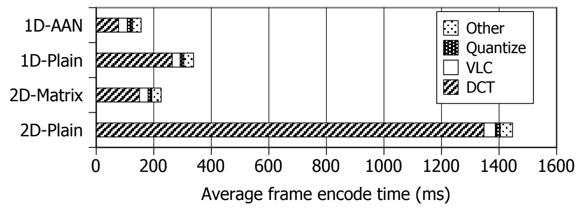


Figure 5: MJPEG encode time on single thread x86

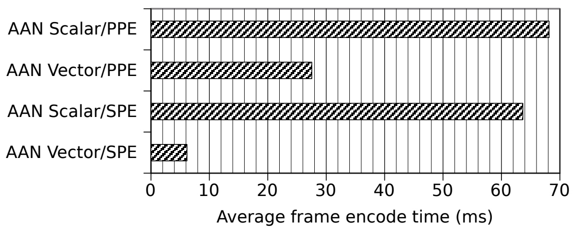


Figure 6: Encoding performance on Cell with different implementations of AAN and VLC placement

the previous encoding speed was limited by the VLC as can be seen in Figure 6 (Scalar- and Vector/SPE).

To get some insight into SPE utilization, we collected a trace (using pdtr, part of IBM SDK for Cell) showing how much time is spent on the encoding parts. Figure 7 shows the SPE utilization when encoding HD frames for the Scalar- and Vector/SPE from Figure 6. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand-code SIMD intrinsics to obtain high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version, and additionally 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

With the vector version of DCT and quantization, the VLC coding uses about 80 % of each SPE. This can possibly be optimized further, but we did not find time to pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to get high performance on this architecture. In particular, correctly implementing an algorithm using vector intrinsics is imperative. Of the 14 implementations for Cell in Figure 1, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization, and the Vector/SPE implementation in Figure 6 is a combination of these two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit-shifting of large fields on Cell as well as GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles, as seen in Figure 5, and the awkward process of gathering profiling data on multicore systems later on. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work, and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are coupled loosely. In that way, they can both be replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 1 to find out what differentiates the fastest from the medium speed implementations, we found some distinguising features: The most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfers and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed, DMA operations that left SPEs in a stalled state waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture, and we have only touched upon a few of them. Some of these are

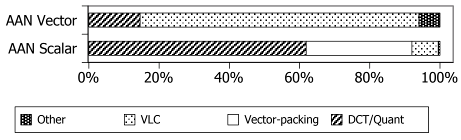


Figure 7: SPE utilization using scalar or vector DCT

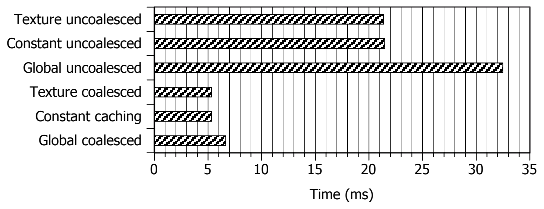


Figure 8: Optimization of GPU memory accesses

obvious, but not all, and to get acceptable performance out of a program running on the Cell architecture may require multiple iterations, restructuring and even rewrites.

3.3 GPU Experiments

As for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Coalesced memory accesses are known to have large performance impacts. However, few quantified results exist, and efficient usage of memory types, alignment and access patterns remains an art. Weimer et al. [11] experimented with bank conflicts in shared memory, but to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and write data to and from memory with both uncoalesced and coalesced access patterns [7], and used the Nvidia CUDA Visual Profiler to isolate the GPU-time for the different kernels.

Figure 8 shows that an uncoalesced access pattern decreases throughput in the order of four times due to the increased number of memory transactions. Constant and texture memory are cached, and the performance for uncoalesced accesses to them is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative even when cached memory types are used. It is also important to make sure the memory accesses are correct according to the specifications of particular GPUs because the optimal access patterns vary between GPU generations.

To find out how memory accesses and other optimizations affect programs like a MJPEG encoder, we experimented with different DCT implementations. Our baseline DCT algorithm is the 2D-plain algorithm. The only optimizations in this implementation are that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. As we observed in Figure 8, cached memory spaces improve performance compared to global memory, especially when memory accesses are uncoalesced. The second implementation, referred to as 2D-plain optimized, is tuned to run efficiently using principles from the CUDA Best Practices Guide [6]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using boolean arithmetics and manual loop unrolling. Our third implementation, the 1D-AAN algorithm, is based upon the scalar implementation

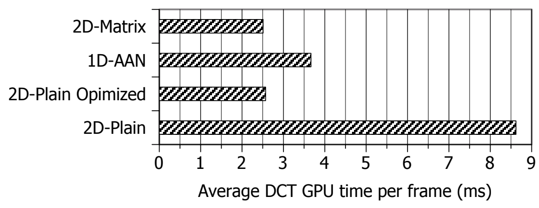


Figure 9: DCT performance on GPU

used on the Cell. Every macroblock is processed with eight threads, one thread per row of eight pixels. The input image is stored in cached texture memory, shared memory is used for temporarily storing data during processing. Finally, the 2D-matrix DCT using matrix multiplications where each matrix element is computed by a thread. The input image is stored in cached texture memory, and shared memory is used for storing data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU were used. Transferring data over the PCI Express bus is expensive, and asynchronous transfers help us reuse the kernels and hides some of the PCI Express latency by transferring data in the background.

To isolate the DCT performance, we used the CUDA Visual Profiler. The profiling results of the different implementations can be seen in Figure 9, and we can observe that the 2D-plain optimized algorithm is faster than AAN. The 2D-plain algorithm requires significantly more computations than the others, but by correctly implementing it, we get almost as good performance as with the 2D-matrix. The AAN algorithm, which does the least amount of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

This experiment shows that for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the theoretical complexity of an algorithm.

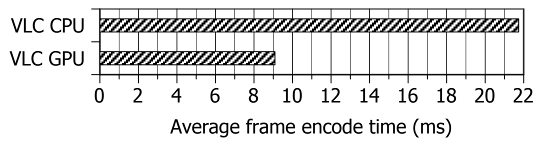


Figure 10: Effect of offloading VLC to the GPU

The last GPU experiment considers entropy coding on the GPU. As for the Cell, VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host, or by using atomic OR on the GPU. For the experiments here, we chose the former since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 10 shows the results of an experiment that compares MJPEG with AAN DCT with VLC performed on the host and on the GPU, respectively. We achieved a doubling of the encoding performance when running VLC on the GPU. In this particular case offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or faster GPU, the encoder will still scale.

4. DISCUSSION

Heterogeneous architectures like Cell and GPU provide large amounts of processing power, and achieving encoding throughputs of 480 MB/s and 465 MB/s, respectively, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions, and that there may be large differences in the way a programmer must think.

The architectures of GPU and Cell are very different, and in this respect, some algorithms may be more suited than others. This can be seen in the experiments, where the AAN algorithm for DCT calculation performed best on both x86 and Cell, but did not achieve the highest throughput on GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

The programming models used on Cell and GPU mandate two different ways of thinking parallel. The approach of Cell is very similar to multi-threaded programming on x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches, and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads, and only synchronization through barriers. This hides the architecture complexity, and is therefore a simpler concept to grasp for some programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figure 1. However, to get the highest possible performance, the programmer must also understand the nitty details of the architecture to avoid pitfalls like warp divergence and uncoalesced memory accesses.

Deciding at which granularity the data should be partitioned is very hard to do correct *a priori*. The best granularity for a given problem differs with the architecture and even different models of the same architecture. One approach towards accomplishing this is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data partitioning, code partitioning is hard to do correctly in advance. In general, a rule of thumb is to write modular code to allow moving the parts to other cores. Also, a fine granularity is beneficial, since small modules can be merged again, and also be executed repeatedly with small overhead. Offloading is by itself advantageous as resources on the main processor become available for other tasks. It also improves scalability of the program with new generations of hardware. In our MJPEG implementations, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance on both Cell and GPU, but it may not always be the case that offloading provides higher throughput.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort for accomplishing this throughput was much higher on the Cell. This was mainly caused by the tedious process of writing a SIMD version of the encoder. Porting the encoder to the GPU in a straight-forward manner without significant optimizations for the architecture yielded a very good offloading performance compared to native x86. This indicates that the GPU is easier to use, but to reap the full potential of the architecture, one must have the same deep level of understanding as with the Cell architecture.

5. RELATED WORK

Heterogeneous multi-core platforms like the Cell and GPUs have attracted a considerable amount of research that aims at optimizing specific applications for the different architectures such as [9] and [4]. However, little work has been done to compare general optimization details of different heterogeneous architectures. Amesfoort et al. [10] have evaluated different multicore platforms for data-intensive kernels. The platforms are evaluated in terms of application performance, programming effort and cost. Colic et al. [3] look at the application of optimizing motion estimation on GPUs and quantify impact of design choices. The workload investigated in this paper is different from the workload we benchmark in our experiments, but they show a similar trend as our GPU experiments. They also conclude that elegant solutions are not easily achievable, and that it takes time, practice and experience to reap the full potential of the architecture. Petrini et al. [8] implement a communication-heavy radiation transport problem on Cell. They conclude that it is a good approach to think about problems in terms of five dimensions and partitioning them into: process parallelism at a very large scale, thread-level parallelism that handles inner loops, data-streaming parallelism that double-buffers data for each loop, vector parallelism that uses SIMD functions within a loop, and pipeline parallelism that overlaps data access with computations by threading. From our MJPEG implementations we observed that programmers had difficulties thinking parallel in two dimensions. This level of multi-dimensional considerations strengthens our statement that intrinsic knowledge of the system is essential to reap full performance of heterogeneous architectures.

6. CONCLUSION

Heterogeneous, multicore architectures like Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial, and in order to learn how to think and use the resources efficiently, we have experimentally evaluated several issues to find the tricks and troubles.

In general, there are some similarities, but the way of thinking must be substantially different - not only compared to an x86 architecture, but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account both when choosing a specific algorithm and making implementation-specific decisions. A lot of trust is put on the compilers of development frameworks and new languages like Open CL, which are supposed to be a "recompile-only" solution. However, to tune performance, the application must still be hand-optimized for different versions of the GPUs and Cells available.

Acknowledgements

The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and National Science Foundation, for the use of Cell Broadband Engine resources. We also acknowledge Alexander Ottesen, Ståle Kristoffersen, Øystein Gyland, Kristoffer Egil Bonarjee, Kjetil Endal and Kristian Evensen for their contributions.

7. REFERENCES

[1] Arai, Y., Agui, T., and Nakajima, M. A fast dct-sq scheme for images. *Transactions of IEICE E71*, 11 (1988).

[2] Cabeen, K., and Gent, P. Image compression and the discrete cosine transform. In *Math 45*, College of the Redwoods.

[3] Colic, A., Kalva, H., and Furht, B. Exploring nvidia-cuda for video coding. In *ACM SIGMM conference on Multimedia systems (MMSys)* (2010), ACM, pp. 13–22.

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

[5] Kovac, M., and Ranganathan, N. JAGUAR: A fully pipelined VLSI architecture for JPEG image compression standard. *Proceedings of the IEEE 83*, 2 (1995).

[6] Nvidia. Nvidia cuda c programming best practices guide 2.3, 2009.

[7] Ottesen, A. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2009.

[8] Petrini, F., Fossuma, G., Fernandez, J., Varbanescu, A. L., Kistler, M., and Perrone, M. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS)* (March 2007), IEEE, pp. 1–10.

[9] Sachdeva, V., Kistler, M., Speight, E., and Tzeng, T.-H. K. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. *Parallel Computing 34*, 11, 616–626.

[10] van Amsesfoort, A., Varbanescu, A., Sips, H. J., and van Nieuwpoort, R. Evaluating multi-core platforms for hpc data-intensive kernels. In *ACM Conference on Computing Frontiers (ICCF)* (2009).

[11] Weimer, W., Boyer, M., and Skadron, K. Automated dynamic analysis of cuda programs. In *Third Workshop on software Tools for MultiCore Systems (STMCS)* (2008).

Paper III: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Title: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data [92].

Authors: H. Espeland, P. Beskow, H. K. Stensland, P. N. Olsen, S. B. Kristoffersen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2011 International Conference on Parallel Processing Workshops, IEEE, 2011.

P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Håvard Espeland, Paul B. Beskow, Håkon K. Stensland, Preben N. Olsen,
Ståle Kristoffersen, Carsten Griwodz, Pål Halvorsen

Department of Informatics, University of Oslo, Norway
Simula Research Laboratory, Norway
Email: {haavares, paulbb, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

***Abstract*—The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason, Google and Microsoft implemented their respective processing frameworks MapReduce [10] and Dryad [19], as they allow the developer to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation in the design of these *batch* processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs, or even pre-determined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution.**

With the Nornir runtime system for parallel programs [39], which is a Kahn Process Network implementation, we addressed and solved several of these limitations. However, it is more difficult to use than other frameworks due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called *P2G*, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines, and provides both data- and task-parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We have implemented an (interchangeable) P2G *kernel language* to ease development. In this paper, we present a proof of concept implementation of a P2G execution node and some experimental examples using complex workloads like Motion JPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

I. Introduction

Live, interactive multimedia services are steadily growing in volume. Interactively refined video search, dynamic participation in video conferencing systems and user-controlled views in live media transmissions are a few examples of features that future consumers will expect when they consume multimedia content. New usage patterns, such as extracting features in pictures to identify objects, calculation of 3D depth information from camera arrays, or generating free-view videos from multiple camera sources in real-time, add further magnitudes of processing requirements to already computationally intensive tasks like traditional video encoding. This fact is further exacerbated by the advent of high-definition videos.

Many-core systems, such as graphic processor units (GPUs), digital signal processors (DSPs) and large scale distributed systems in general, provide the required processing power, but taking advantage of the parallel computational capacity of such hardware is much more complex than single-core solutions. In addition, heterogeneous hardware requires individual adaptation of the code, and often involve domain specific knowledge. All this places additional burdens on the application developer. As a consequence, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [10] and Microsoft's Dryad [19]. These frameworks are limited by their design for batch processing of large amounts of data, with few dependencies across a large cluster of machines. Modifications and enhancements that address bottlenecks [8] together with support for new types of workloads and additional hardware exist [9], [16], [31]. It is also worth mentioning that new languages for current batch frameworks have been proposed [29], [30]. However, the development and processing of distributed multimedia applications is inherently more difficult. Multimedia applications also have stricter requirements for flexibility. Support for iterations is essential, and knowledge of deadlines is often imperative. The traditional batch processing frameworks do not support this.

In our Nornir runtime system for parallel processing [39], we addressed many of the shortcomings of the batch processing frameworks. Nornir is based on the idea of Kahn Process Networks (KPN). Compared to MapReduce-like approaches, Nornir adds support for arbitrary processing graphs, deterministic execution, etc. However, KPNs are designed with some unrealistic assumptions (like unlimited queue sizes), and the Nornir programming model is much more complex than that of frameworks like MapReduce and Dryad. It demands that the application developer establishes communication channels manually to form the dependency graph.

In this paper, we expand on our visions and present our initial ideas of *P2G*. It is a completely new framework for distributed real-time multimedia processing. P2G is designed to work on continuous flows of data, such as live video streams, while still maintaining the ability to support batch workloads. We discuss the initial ideas and present a proof-of-concept prototype[1] running on x86 multi-core machines. We present experimental results using concrete multimedia examples. Our main conclusion is that the P2G approach is a step in the right direction for development and execution of complex parallel workloads.

II. Related Work

A lot of research has been dedicated to addressing the challenges introduced by parallel and distributed programming. This has led to the development of a number of tools, programming languages and frameworks to ease the development effort.

For example, several solutions have emerged for simplifying distributed processing of large quantities of data. We have already mentioned Google's MapReduce [10] and Microsoft's Dryad [19]. In addition, you have IBM's System S and accompanying programming language SPADE [13]. Yahoo have also implemented a programming language with their PigLatin language [29], other notable mentions for increased language support is Cosmos [26], Scope [6], CIEL [25], SNAPPLE [40] and DryadLINQ [41]. The high-level languages provide easy abstractions for the developers in an environment where mistakes are hard to correct.

Dryad, Cosmos and System S have many properties in common. They all use directed graphs to model computations and execute them on a cluster. System S also supports cycles in graphs, while Dryad supports non-deterministic constructs. However, not much is known about these systems, since no open implementations are freely available. MapReduce on the other hand has become one of the most cited paradigms for expressing parallel computations. While Dryad and System S use a task parallel model, MapReduce uses a data-parallel model based on keys and values. There are several implementations of MapReduce for clusters [1], multi-core [31], the Cell BE architecture [9], and also for GPUs [16]. Map-Reduce-Merge [8] adds a merge step to process data relationships among heterogeneous data sets efficiently, operations not directly supported by the original MapReduce model. In Oivos [35], the same issues are addressed, but in addition, this system provides a more expressive, declarative programming model. Finally, reducing the layering overhead of software running on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

An inherent limitation in MapReduce, Dryad and Cosmos is their inability to model iterative algorithms. In addition, the rigid MapReduce semantics do not map well to all types of problems [8], which may lead to unnaturally expressed solutions and decreased performance [38]. The limited support for iterative algorithms has been mitigated in HaLoop [5], a fork of Hadoop optimized for batch processing of iterative

[1]The P2G source code and workload examples are available for download from http://www.p2gproject.org/.

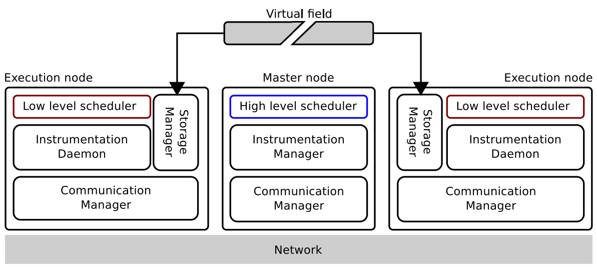


Figure 1. Overview of nodes in the P2G system.

algorithms where data is kept local for future iterations of the MR steps. However, the programming model of MapReduce is designed for batch processing huge datasets, and not well suited for multimedia algorithms. Finally, Google's patent on MapReduce [11] may prompt commercial actors to look for an alternative framework.

KPN-based frameworks are one such alternative. KPNs support arbitrary communication graphs with cycles and are deterministic. However, in practice, very few general-purpose KPN runtime implementations exist. Known implementations include the Sesame project [34], the process network framework [28], YAPI [22] and our own Nornir [39]. These frameworks have several benefits, but for application developers, the KPN model has some challenges, particularly in a distributed scenario. To mention some issues, a distributed version of a KPN implementation requires a distributed deadlock detection and a developer must specify communication channels between the processes manually.

An alternative framework based on a process network paradigm is StreamIt [15], which comprises a language and a runtime system for simplifying the implementation of stream programs described by a graph that consists of computational blocks (filters) with a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [23]. The compiler produces code that can make use of multiple machines or CPUs, whose number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications. Multimedia applications have strict requirements and knowledge of deadlines is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resouces change. Thus, all of the existing frameworks have some short-comings that are difficult to address, and the traditional batch processing frameworks simply come up short in our multimedia scenario. Next, inspired by the strengths of the different approach, we present our ideas for a new framework for distributed real-time multimedia processing.

III. Basic idea

The idea of P2G was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict runtime processing to the domain of data parallelism [12]. Functional languages such as Erlang [3] and Haskell [18] and the event-based SDL [21], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels.

In our multimedia scenario, Nornir improves on many of the shortcomings of the traditional batch processing frameworks, like MapReduce and Dryad. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles/iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is task-parallel, and data-parallelism must be explicitly added by the programmer. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, having exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations, however, queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

With P2G, we build on the knowledge gained from developing Nornir and address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature for processing multimedia workloads includes automatic combined task and data parallelism. Intra-frame prediction in H.264 AVC, for example, introduces many dependencies between sub-blocks of a frame, and together with other overlapping processing stages, these operations have a high potential for benefiting from both types of parallelism. We demonstrated the potential in earlier work with Nornir, whose deterministic nature showed great parallelization potential in processing arbitrary dependency graphs.

Multimedia algorithms being iterative by nature exhibit many pipeline parallel opportunities. Exploiting them are hard because intrinsic knowledge of fine-grained dependences are required, and structuring programs in such a way that pipeline parallelism can be used is difficult. Thies et al. [33] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses assuming that the behaviour is stable. They evaluated their system on multimedia algorithms and gained significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, and this enable the runtime to automatically detect and take full advantage of all parallel opportunities without manual intervention.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. The source of this non-deterministic behavior can be removed by adopting strict write-once semantics for writing to memory [4]. Languages that take advantage of the concept of single assignment include Erlang [3] and Haskell [18]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once-semantics are well-suited for a scheduler's dependency analysis, it is not straight-forward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [20] and SIFT [24] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without loosing write-once-semantics.

Flexible partitioning requires the processing of clearly distinct data units without side-effects. The idea adopted for P2G is to use *kernels* as in stream processing [15], [27]. Such a kernel is written once and describes the transformation of multi-dimensional fields of data. Where such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which can analyze dependencies and guarantee fully deterministic output independent of order due to the write-once semantics of fields.

Together, these observations determined four basic ideas for the design of P2G:

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.

- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

Within the boundaries of these basic ideas, P2G should be easily accessible for programmers who only need to write isolated, sequential pieces of code embedded in kernel definitions. The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to fetch slices of a field in as fine a granularity as possible, supporting data parallelism.

P2G is designed to be language independent, however, we have defined a C-like language that captures many of P2G's central concepts. As such, the P2G language is inspired by many existing languages. In fact, Cray's Chapel [7] language antedates many of P2G's features in a more complete manner. P2G adds, however, write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements that formulate data dependencies between implicitly instantiated kernels, which are (currently) written in C/C++.

The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, P2G supports deadlines, which allows scheduling decisions such as termination, branching and the use of alternative code paths based on runtime observations.

In summary, we have opted for an idea that allows programmers to focus on data transformations in a sequential manner, while simultaneously providing enough information for dynamically adapting the data and task parallelization. As an end result of our considerations. P2G's fields look mostly like global multi-dimensional arrays in C, although their representation in memory may deviate, i.e., they need not be placed contiguously in the memory of a single node, and may even be distributed across multiple machines. Although this looks contrary to our message-based KPN approach used in Nornir, it maps well when slices of fields are interpreted as messages and the run-queues of worker threads as KPN channels. An obvious difference is that fields can be read as often as necessary.

IV. Architecture

As shown in figure 1, the P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (a graph of multi-core and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. As such, the global topology can change during runtime as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses a two-level scheduling approach. On the master node, we have a high-level scheduler (HLS), and on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workloads store and fetch statements, from which it can generate an intermediate implicit static dependency graph (see figure 2) where edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field (as seen in figure 3). From the intermediate graph, the HLS can then derive a final implicit static dependency graph (see figure 3). The HLS can then use a graph partitioning [17] or search based [14] algorithm to partition the workload into a suitable number of components that can be distributed to, and run, on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload the final graph can be weighted with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of improving the throughput in the system, or accommodate for changes in the global load.

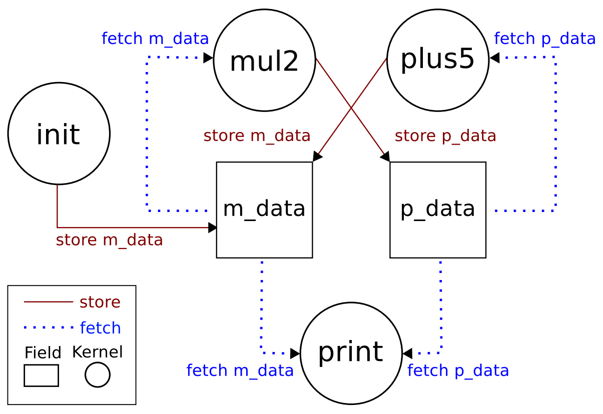


Figure 2. Intermediate implicit static dependency graph

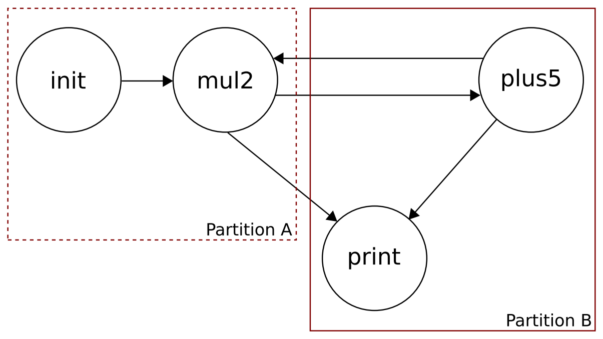


Figure 3. Final implicit static dependency graph

Given a partial workload (such as *partition A* from figure 3), an LLS at an execution node is responsible for maximizing local scheduling decisions. We discuss this further in section V, but figure 4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G, and take advantage of specialized hardware, such as GPUs.

This idea of using a two level scheduling approach is not new. It has also been considered by Roh et al. [32], where they have performed simulations on parallel scheduling decisions for instruction sets of a functional language. Simple workloads are mapped to various simulated architectures, using a "merge-up" algorithm, which is equivalent to our LLS, and "merge-down" algorithm, which is equivalent to our HLS. These algorithms cluster instructions in such a way that parallelism is not limited, their conclusion is that utilizing a merge-down strategy often is better.

Data distribution, reporting, and other communication patterns is achieved in P2G through an event-based, distributed publish-subscribe model. Dependencies between components in a workload are deterministically derived from the code and the high-level schedulers partitioning decisions, and direct communication occurs.

As such, P2G relies on its combination of a HLS, LLS, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

V. Programming model

The programming model of P2G consists of two central concepts, the *implicit static dependency graph* (figures 2 and 3) and the *dynamically created directed acyclic dependency graph* (DC-DAG) (figure 4). We have also developed a *kernel language* (see figure 5), to make it easier to develop applications using the P2G programming model, though we consider this language to be interchangeable.

The example we use throughout this discussion consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiples a value by 2 and stores this data, which *plus5* then fetches and increases by 5, *mul2* then fetches the data stored by *plus5*, and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they have produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes some initial data for *mul2* to consume. The kernels operate on two 1-dimensional, 5 element fields. The print kernel writes *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* for the first *age* and *{25, 27, 29, 31, 33}*, *{50, 54, 58, 62, 66}* for the second, etc (as seen in figure 4). As such, the first *iteration* produces the data: *{10, 11, 12, 13, 14}*, *{20, 22, 24, 26, 28}* and *{25, 27, 29, 31, 33}*, and the second *iteration* produces the data: *{50, 54, 58, 62, 66}* and *{55, 59, 63, 67, 71}*, etc. Since there is no termination condition for this program it runs indefinitely.

A. Dependency graphs

The intermediate implicit static dependency graph (as seen in figure 2) is derived from the interaction between fields and kernel definitions, more precisely from the *fetch* and *store* statements of a kernel definition. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in figure 3. This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. The graph can be further weighted using instrumentation data, to serve as input for repartitioning. It is important to note that these weighted graphs can serve as input to static offline analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload, given various global topology configurations.

During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as seen in figure 4. This expansion from a cyclic graph to a directed acyclic graph occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iteration. We have chosen to call each such unrolled loop an *Age*. The LLS can then use the DC-DAG to combine tasks and data to reduce overhead introduced by P2G and to take advantage of specialized hardware, such as GPUs. It can then try different combinations of these low-level scheduling decisions to improve the throughput of the system.

We can see how this is accomplished in figure 4. When moving from *Age=1* to *Age=2*, we can see how the LLS has made a decision to reduce data parallelity. In P2G, kernels fetch slices of data, and initially *mul2* was defined to work on each single field entry in parallel, but in *Age=2*, the LLS has decreased the granularity of the fetch statement to encompass the entire field. It could also have split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field.

Moving from *Age=2* to *Age=3*, we see how the LLS has made a decision to decrease the task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data has been fully processed by each task. If the *print* kernel was not present, storing to the intermediate field *m_data* could be circumvented in its entirety.

Finally, moving from *Age=3* to *Age=4*, we can see how a decision to decrease both task and data parallelity has been taken. This renders this single kernel instance effectively instance into a classical *for-loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the data.

P2G makes runtime adjustments dynamically to both data and task parallelism based on the possibly oscillating resource availability and the reported performance monitoring.

B. Kernel language

From our experience with developing Nornir, we came to the realization that expressing workloads in a framework capable of supporting such complex graphs without a high-level language is a difficult task. We have therefore developed a *kernel language*. An implementation of a simple workload is outlined in figure 5, with a C++ equivalent listed in figure 6.

In the current version of our system, P2G is exposed to the developer through this *kernel language*. The language itself is not an integral part and can be replaced easily. However, it exposes several foundations of the P2G design. Most important are the kernel and field definitions, which describe the code and interaction patterns in P2G.

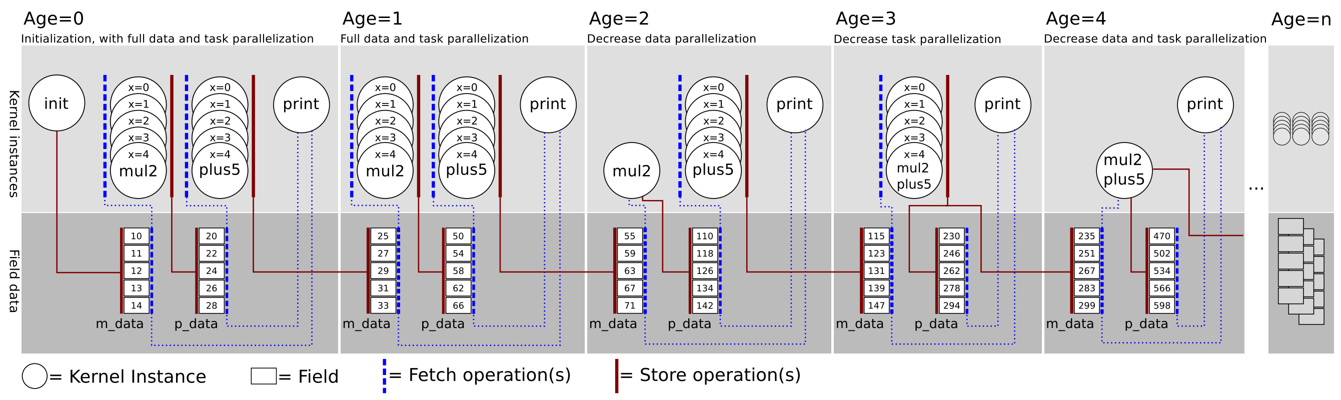
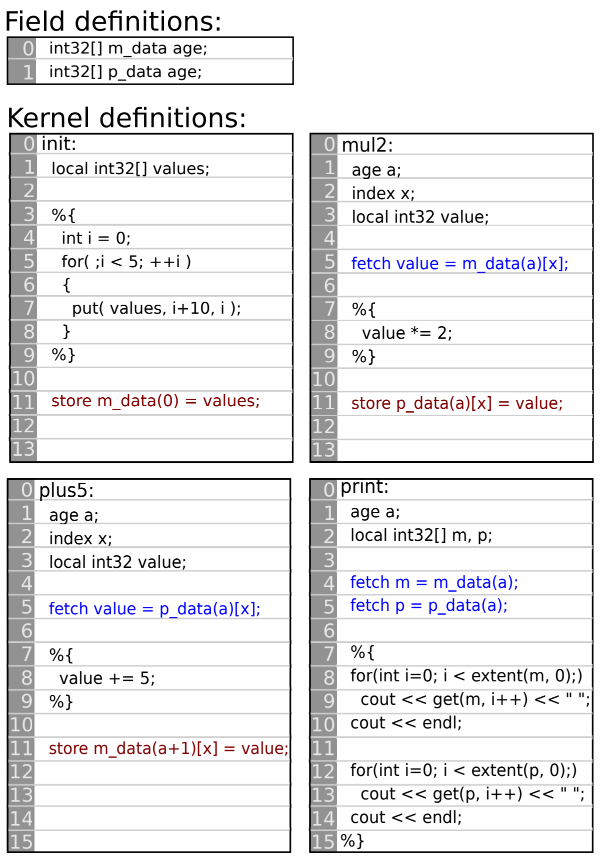


Figure 4. Dynamically created directed acyclic dependency graph (DC-DAG)



Field definitions:

```
int32[] m_data age;
int32[] p_data age;
```

Kernel definitions:

```
init:
 local int32[] values;

 %{
  int i = 0;
  for( ;i < 5; ++i )
  {
   put( values, i+10, i );
  }
 %}

 store m_data(0) = values;
```

```
mul2:
 age a;
 index x;
 local int32 value;

 fetch value = m_data(a)[x];

 %{
  value *= 2;
 %}

 store p_data(a)[x] = value;
```

```
plus5:
 age a;
 index x;
 local int32 value;

 fetch value = p_data(a)[x];

 %{
  value += 5;
 %}

 store m_data(a+1)[x] = value;
```

```
print:
 age a;
 local int32[] m, p;

 fetch m = m_data(a);
 fetch p = p_data(a);

 %{
 for(int i=0; i < extent(m, 0);)
  cout << get(m, i++) << " ";
 cout << endl;

 for(int i=0; i < extent(p, 0);)
  cout << get(p, i++) << " ";
 cout << endl;
%}
```

Figure 5. Kernel and field definitions

A kernel definition's primary purpose is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. As such, a field serves as an interaction point for kernel definitions, as can be seen in figure 2.

An important aspect of multimedia workloads is the ability to express deadlines, where it does not make sense to encode a frame if the playback has moved past that point in the

```
void print( int *data, int num )
{
    for( int i = 0; i < num; ++i )
        std::cout << data[i] << "␣";
    std::cout << std::endl;
}
int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    while( true )
    {
        for( int i = 0; i < 5; ++i )
            p_data[i] = m_data[i] * 2;

        print( m_data, 5 );
        print( p_data, 5 );

        for( int i = 0; i < 5; ++i )
            m_data[i] = p_data[i] + 5;
    }
    return 0;
}
```

Figure 6. C++ equivalent of mul/sum example

video-stream. Consequently, we have implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer: *timer t1*. This timer can then be polled, and updated, from within a kernel definition, for example *t1+100ms* or *t1 = now*. Given a condition based on a deadline such as *t1+100ms*, a timeout can occur and an alternate code-path can be executed. Such an alternate code-path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. Currently, we have basic support for expressing deadlines in the kernel language, but the semantics of these expressions require refinement, as their implications can be considerable.

Fields in P2G have a number of properties, including a type and a dimensionality. Another property is, as mentioned above, *aging*, which allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times, as long as the age increases for each store operation (as seen in figure 4). In essence, this adds a dimension to the field and makes it possible to accommodate iterative algorithms. Additionally, it is important to realize that fields are not connected to any single node, and can be fully localized or distributed across multiple execution nodes (as seen in figure 1).

In defining the interaction between kernels and fields, it is encouraged that the programmer expresses the finest possible granularity of kernel definitions, and, likewise, the most precise slices possible for the kernel within the field. This is encouraged because it provides the low-level scheduler more control over the granularity of task and data decomposition. Aided by instrumentation data, it can reduce scheduling overhead by combining several instances of a kernel that process different data, or several instances of different kernels that process data in sequence (as seen in figure 4). The scheduler makes its decisions based on the implicit static dependency graph and instrumentation data.

C. Runtime

Following from the previous discussions, we can extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime, and the number of kernel instances executed in parallel for a given kernel definition depends on its fetch statements.

To clarify, a kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements of the kernel definition. For example, looking at figure 4 and 5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x* fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum possible *x* kernel instances simultaneously per age, giving *a*x mul2* kernel instances. Though, as we have seen, this number can be decreased by the scheduler making *mul2* work over larger slices of data from *m_data*.

With P2G we support implicit resizing of fields, this can be witnessed by looking at the kernel definition of *print* in figure 5. Initially, the extents of *m_data* and *p_data* are not defined, as such, with each iteration of the *for*-loop in *init* the local field *values* is resized locally, leading to a resize of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resize, such as *p_data*. Following the discussion from the previous paragraph, such an implicit resize can lead to additional kernel instances being dispatched.

It is worth noting that a kernel instance is only dispatched when all its dependencies are fulfilled, i.e., that the data it fetches has been stored to the respective fields and elements. Looking at figure 4 and 5 again, we can see that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, this means that the kernel instance *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. In our system, each kernel instance is only dispatched once, due to our write-once semantics. To summarize, the *print* kernel instance working on *age=0* becomes runnable when all the elements of *m_data* and *p_data* for *age=0* have been stored. Once it has become runnable, it is dispatched and runs only once.

VI. Prototype Implementation

To verify the feasibility of the P2G framework presented in this paper, we have implemented a prototype version. The prototype consists of a compiler for the kernel language and a runtime that can execute P2G programs on multi-core linux machines.

A. Compiler

Programs written for the P2G system are designed to be platform independent and feature native blocks of code written in C or C++. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as nVIDIA's nvcc compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. We decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control of the resulting object code, but we gain the flexibility and sophisticated optimization of the native compilers, resulting in a lightweight P2G compiler. The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

B. Runtime

The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors using the implicit dependency graph formed by kernel definitions. However, at the time of writing, the prototype runtime does not yet have a full implementation of deadline expressions, this is because the semantics of the kernel language support for this feature is not fully defined yet.

The prototype targets a node with multiple processors. It is designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which may require resize operations. A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernels *fetch* statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement, and puts these in a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

The low-level scheduler consists of a dependency analyzer and kernel instance dispatcher. Using the implicit dependency graph, the dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker

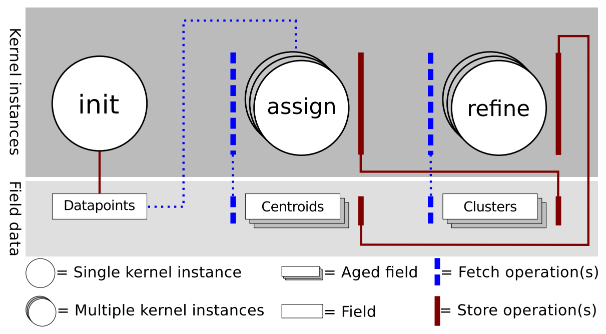


Figure 7. Overview of the *K*-means clustering algorithm

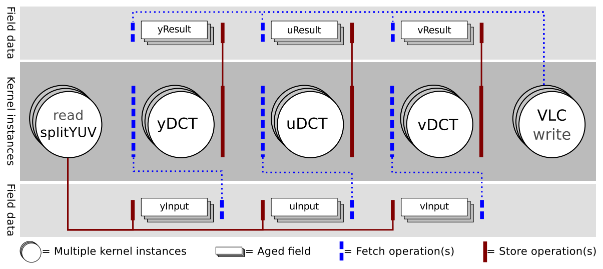


Figure 8. Overview of the MJPEG encoding process

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table I
OVERVIEW OF TEST MACHINES

threads. Dependencies are analyzed in a dedicated thread which handles events emitted from running kernel instances that notifies on *store* and *resize* operations performed on fields. Kernel instances are executed by a worker thread dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements or by groups of kernels that satisfy their own dependencies in aging cycles, such as the *mul2* and *plus5* kernel in figure 5.

The runtime is written in C++ and uses the blitz++ [37] library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://www.p2gproject.org/.

VII. Workloads

We have implemented a few workloads commonly used in multimedia processing to test the prototype implementation. The P2G kernel language is able to expose both the data and task parallelism of the programs to the P2G system, so the runtime is able to adapt execution of the programs to suit the target architecture.

A. K-means clustering

K-means clustering is an iterative algorithm for cluster analysis which aims to partition n datapoints into k clusters in which each datapoint belongs to the cluster with the nearest mean. As shown in figure 7, the P2G k-means implementation consists of an *init* kernel, which generates n datapoints and *stores* them to the datapoints field. Then, it selects k of these datapoints randomly, as the initial means, and *stores* them to the centroids field. Next, the *assign* kernel *fetches* a slice of data, a single datapoint per *kernel instance*, the last calculated centroids, and *stores* this datapoint to the cluster of the closest centroids using the euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* form a loop which gradually leads to a convergence in centroids, at which point the k-means algorithm has completed.

B. Motion JPEG

Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images. The MJPEG format provides many layers of parallelism, well suited for illustrating the potential of the framework. We focused on optimizing the discrete cosine transform (DCT) and quantization part as this is the most compute-intensive part of the codec.

The *read* + *splitYUV* kernel reads the input video in YUV-format and stores the data in three global fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 8, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC* + *write* kernel store the MJPEG bit-stream to disk.

VIII. Evaluation

We have run tests with the workloads Motion JPEG and *K*-means (described in section VII). Each test was run on a *4-way Core i7* and an *8-way Opteron* (see table I for hardware specifications) ranging from *1* worker thread to *8* worker threads with *10* iterations per worker thread count. The results of these tests are reported in the figures 10 and 9, which show the mean running time in *seconds* for each machine for a given thread count with standard deviation reported as error-bars.

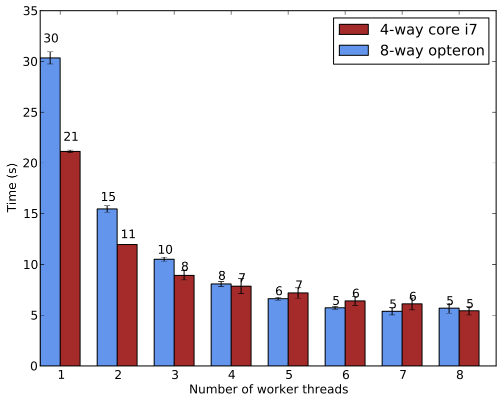


Figure 9. Workload execution time for Motion JPEG

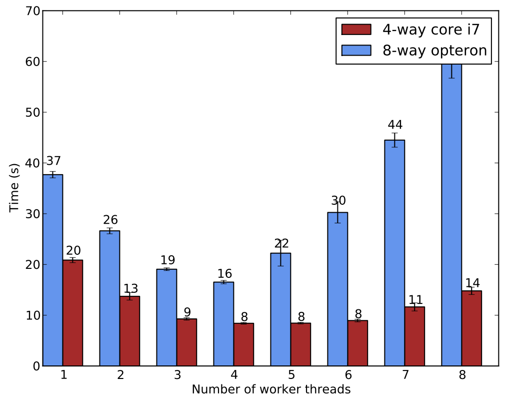


Figure 10. Workload execution time for K-means

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table II
MICRO-BENCHMARK OF MJPEG ENCODING IN P2G

In addition, we have performed micro-benchmarks for each workload, summarized in the tables II and III. The benchmarks summarize the number of kernel instances dispatched per kernel definition, dispatch overhead and time spent in kernel code.

A. Motion JPEG

The Motion JPEG workload is run on the standard test sequence *Foreman* encoded in *CIF* resolution. We limited the workload to process *50* frames of video.

As we can observe from figure 9, P2G is able to scale close to linearly with the resources it has available. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the running time when moving from *7* to *8* worker threads. Where the eighth thread shares resources with the dependency analyzer. To compare, the standalone single threaded MJPEG encoder on which the P2G version is based upon has a running time of *30* seconds on the Opteron machine and *19* seconds on the Core i7 machine. Note that both the standalone and P2G versions of the MJPEG encoder use a naive DCT calculation, there are versions of DCT that can significantly improve performance, such as FastDCT [2].

From table II, we can see that time spent in kernel code is considerably higher compared to the dispatch overhead for the kernel definitions. The dispatch time includes allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have a considerably higher dispatch time then the **DCT* operations.

We can also see that the majority of CPU-time is spent in the kernel instances of *yDCT, uDCT* and *vDCT*, which is the computationally intensive part of the workload. This indicates that decreasing data and task granularity, as discussed in section V-A, has little impact on the throughput of the system. This is because the majority of time is already spent in kernel code.

Note that even though there are *51* instances of the *read-/write* kernel definitions, only *50* frames are encoded, because the last instance reaches the end of the video stream.

B. K-means

The K-means workload is run with K=*100* using a randomly generated data set containing *2000* datapoints. The K-means algorithm is not run until convergence, but with *10* iterations. If we do not define this break-point it is undefined when the algorithm converges, and as such, we have introduced this condition to ensure that we get a relatively stable running time for each run.

As seen in figure 10, the K-means workload scales to *4* worker threads. After this, the running time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition, as witnessed when comparing the dispatch time to the time spent in kernel code. This leads to the serial dependency analyzer becoming a bottle-neck in the system. As discussed in section V-A, this condition could be alleviated by decreasing the granularity of data-parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time and reduce the workload of the dependency analyzer. The reduction in work for the dependency analyzer is a result of the lower number of kernel instances being run.

The two different test machines behave somewhat differently in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks, and we think this is why the Core i7 suffers less when we meet the limitations dictated by Amdahl's law.

The considerable time *init* spends in kernel code is because it generates the data set.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table III
MICRO-BENCHMARK OF K-MEANS IN P2G

C. Summary

We have shown that our prototype implementation of an execution node is able to scale with the available resources, as seen in figure 9 and 10. Our initial results indicate that the functionality of decreasing the granularity of task and data parallelity, as discussed in section V-A, is important to ensure full resource utilization.

IX. DISCUSSION

Even though support for deadlines is not yet fully implemented in the P2G runtime, the concept of deadlines formed an integral part of our design goal. The intention behind deadlines is to accommodate for live multimedia workloads, where real-time requirements are mission essential. Varying conditions over time, both in the workload and topology, may effect scheduling decisions: such as termination, branching and the use of alternative code paths based on runtime observations. This is similar to SDL, but unlike contemporary high performance languages.

In P2G, we encourage the programmer to describe the workload in as fine granularity as possible, both in the functional and data decomposition domains. The low-level scheduler has an understanding of both decomposition domains and deadlines. Given this information, the low-level scheduler can minimize overhead by combining functional components and slices of data by adapting to its available resources, be it local cores, or even GPU execution units.

Write-once semantics on fields incurs a large penalty if implemented naively, both in terms of memory usage and data cache misses. However, as the fields are virtual and do not even have to reside in continuous memory, the compiler and runtime are free to optimize field usage. This includes re-using buffers for increased cache locality when old ages are no longer referenced, and garbage collecting old ages. The explicit programming model of P2G allows the system to anticipate what data is needed in the future, which can be used for further optimizations.

Given the complexity of multimedia workloads and the (potentially) heterogeneous resources available in a modern topology, and in many cases, no knowledge of the underlying capabilities of the resources (which is common in modern cloud services), mapping these complex multimedia workloads manually to the available resources becomes an increasingly difficult task, and at some point, even impossible. This is particularly the case where resource availability fluctuates, such as in modern virtual machine parks. With batch processing, where the workloads frequently are not associated with some intrinsic deadline, this task is solved, with frameworks such as MapReduce and Dryad. However, for processing continuous streams such as iterative multimedia algorithms in an elastic manner requires new frameworks; P2G is a step in that direction.

X. CONCLUSION

With P2G, we have proposed a new flexible framework for automatic parallel, real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of data and task decomposition. Using our kernel language this decomposition is expressed through kernel definitions and fetch and store statements on fields. This language is independent from the P2G runtime and can easily be replaced. Given a workload defined in our kernel language it is compiled for execution in P2G. This workload can then be partitioned by the high-level scheduler of a P2G master node, which then distributes partitions to P2G execution nodes which runs the tasks locally. Execution nodes can consist of heterogeneous resources. A low-level scheduler at the execution nodes then adapted the partial (or full) workload to run optimally using resources at hand. Feedback from the instrumentation daemon at the execution node can lead to repartitioning of the workload (a task performed by the high-level scheduler). The aim is to bring the ease of batch-processing frameworks to multimedia workloads.

In this paper we have presented an execution node capable of running on a multi-way architecture. This results from our experiments running on this prototype show the potential of our ideas. However, there still remains a number of vectors for optimization. In the low-level scheduler we have identified that combining task and data to minimize overhead introduced by P2G is a first reasonable modification. Additionally, completing the implementation of a fully distributed version is in the pipeline. Also, writing workloads for heterogeneous processing cores like GPUs and non-cache coherent architectures like Intel's SCC is a further consideration. Currently, we are investigating appropriate mechanisms for both high- and low-level scheduling, garbage collection, fat binaries, resource profiling and monitoring, and efficient migration of tasks.

While a number of optimizations remain, we have determined that P2G is feasible, through the implementation of this execution node, and the successful implementation of multimedia workloads, such as Motion JPEG and k-means. With these workloads we have shown that it is possible to express multimedia workloads in the kernel language and we have implemented a prototype of an execution node in the P2G framework that is able to execute kernels and scales with the available resources.

REFERENCES

[1] Apache. Hadoop, Accessed July 2010. http://hadoop.apache.org.

[2] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of IEICE*, E71(11), 1988.

[3] J. Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1–6:26, 2007.

[4] R. Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, 1989.

[5] Y. Blu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI*, pages 10–10, 2004.

[11] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. *US Patent Application*, (US 7650331), 2010.

[12] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[14] F. Glover. Tabu search, Part I1. *ORSA journal on Computing*, 2(1):4–32, 1990.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of PACT*, pages 260–269, New York, NY, USA, 2008. ACM.

[17] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing* 1. *Parallel Computing*, 26(12):1519–1534, 2000.

[18] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1–12:55, 2007.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, New York, NY, USA, 2007. ACM.

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

[21] ITU. *Z.100*, 2007. Specification and Description Language (SDL).

[22] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000*, pages 402–405. ACM Press, 2000.

[23] T. Lee, E.A.; Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[24] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[25] D. Murray, M. Schwarzkopf, and C. Smowton. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[26] C. Nicolaou. An architecture for real-time multimedia communication systems. *Selected Areas in Communications, IEEE Journal on*, 8(3):391–400, 1990.

[27] Nvidia. Nvidia cuda programming guide 3.2, Aug. 2010.

[28] A. G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP*, pages 73–76, 2006.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[30] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of IEEE HPCA*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Roh, W. A. Najjar, and A. P. W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *ACM FPCA: Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1993. ACM.

[33] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. Thompson and A. Pimentel. Towards multi-application workload modeling in sesame for system-level design space exploration. In S. Vassiliadis, M. Berekovic, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[35] S. V. Valvåg and D. Johansen. Oivos: Simple and efficient distributed data processing. In *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 113–122, 2008.

[36] S. V. Valvåg and D. Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *Proc. of IFIP International Conference on Network and Parallel Computing Workshops (NPC)*, pages 174–181, 2009.

[37] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 223–230, London, UK, 1998. Springer-Verlag.

[38] Ž. Vrba, P. Halvorsen, C. Griwodz, and P. Beskow. Kahn process networks are a flexible alternative to mapreduce. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:154–162, 2009.

[39] Ž. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen. The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce. *Journal of Sumpercomputing*, 27(1), 2010.

[40] D. Waddington, C. Tian, and K. Sivaramakrishnan. Scalable lightweight task management for mimd processors, 2011.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-puropose distributed data-parallel computing using a high-level language, 2008.

Paper IV: Low-level Scheduling Implications for Data-intensive Cyclic Workloads on Modern Microarchitectures

Title: Low-level Scheduling Implications for Data-intensive Cyclic Workloads on Modern Microarchitectures [16].

Authors: H. Espeland, P. N. Olsen, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2012 International Conference on Parallel Processing Workshops (ICPPW), IEEE, 2012.

Low-level Scheduling Implications for Data-intensive Cyclic Workloads on Modern Microarchitectures

Håvard Espeland, Preben N. Olsen, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory, Norway   IFI, University of Oslo, Norway
{haavares, prebenno, paalh, griff}@ifi.uio.no

***Abstract*—Processing data intensive multimedia workloads is challenging, and scheduling and resource management are vitally important for the best possible utilization of machine resources. In earlier work, we have used work-stealing, which is frequently used today, and proposed improvements. We found already then that no singular work-stealing variant is ideally suited for all workloads. Therefore, we investigate in more detail in this paper how workloads consisting of various multimedia filter sequences should be scheduled on a variety of modern processor architectures to maximize performance. Our results show that a low-level scheduler additionally cannot achieve optimal performance without taking the specific micro-architecture, the placement of dependent tasks and cache sizes into account. These details are not generally available for application developers and they differ between deployments. Our proposal is therefore to use performance monitoring and dynamic adaption for the cyclic workloads of our target multimedia scenario, where operations are repeated cyclically on a stream of data.**

I. Introduction

There is an ever-growing demand for processing resources, and the trend is to move large parallel and distributed computations to huge data centers or cloud computing. For example, Internet users uploaded one hour of video to YouTube every second in January 2012 [19], an increase of 25% in the last 8 months. Each of these is encoded using several filters that are arranged as vertices in dependency graphs and where each filter implements a particular algorithm representing one stage of a processing pipeline. In our research, we focus on utilizing available resources in the best possible manner for this kind of time-dependent cyclic workloads. The kind of workload is typical for multimedia processing, where large amounts of data are processed through various filters organized in a data-intensive processing pipeline (or graph). Such workloads are supported by for example the OpenCV framework [4].

To process large data sets in general, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [6], IBM's System S [11] and Microsoft's Dryad [13]. However, these frameworks are limited by their design for batch processing with few dependencies across a large cluster of machines. We are therefore currently working on a framework aimed for distributed real-time multimedia processing called P2G [7]. In this work, we have identified several challenges with respect to low level scheduling. The de facto standard is a variant of *work-stealing scheduling* [2], for which we have earlier proposed modifications [17]. We see challenges that have not been addressed by this, and in our work of designing an efficient scheduler, we investigate in this paper how to structure parallel execution of multimedia filters to maximize the performance on several modern architectures.

Performance implications for scheduling decisions on various microarchitectures have been studied for a long time. Moreover, since the architectures are constantly being revised, scheduling decisions that were preferred in the past can harm performance on later generations or competing microarchitectures. We look at implications for four current microarchitectures and how order of execution affects performance. Of recent work, Kazempour et al. [14] looked at performance implications for cache affinity on Clowertown generation processors. It differs from the latest microarchitectures by only having two layers of cache, where only the 32 KB L1 D-cache is private to the cores on a chip multiprocessor (CMP). In their results, affinity had no effect on performance on a single chip since reloading L1 is cheap. When using multiple CMPs, on the other hand, they found significant differences meriting affinity awareness. With the latest generation CMPs having significantly larger private caches (e.g., 256 KB on Sandy Bridge, 2 MB on Bulldozer), we can expect different behavior than on Clowertown. In terms of schedulers that take advantage of cache affinity, several improvements have been proposed to the Work Stealing model. Acar et al. [1] have shown that the randomized stealing of tasks is cache unfriendly and suggest a model that prefers stealing tasks where the worker thread has affinity with that task and gains increased performance.

In this paper, we present how a processing pipeline of real-world multimedia filters runs on state-of-the-art processors. That there are large performance differences between different architectures is to be expected, but we found so large differences between modern microarchitectures even within the same family of computers (x86) making it hard to make scheduling decisions for efficient execution. For example, our experiments shows that there are huge gains to be earned looking into cache usage and task dependencies. There are also huge differences in the configuration of the processing stages, e.g., when changing the amount of rotation in an image, giving completely different resource requirements. Based our these observations, it is obvious that the low-level scheduling should not only depend on the processing pipeline with the dependencies between tasks, but also the specific micro-architecture and the size of the caches. We discuss why scheduling approaches such as standard work stealing models do not result in an optimal performance, and we try to give some insights that a future scheduler should follow.

II. Design and Implementation

Inspired by prevalent execution systems such as StreamIT [16] and Cilk [3], we look at ways to execute data-intensive streaming media workloads better and examine how different processing schemes affect performance. We also want to investigate how these behave on different processors. Because we are processing continous data streams, we are not able to exploit task parallelism, e.g., by processing independent frames of a video in parallel and therefore seek to parallelize within the data domain. A scenario with embarrassingly parallel workloads, i.e., where every data element can be processed independently and without synchronization, is video stream processing. We believe that processing a continous flow of video frames is a reasonable example for several embarrassingly parallel data bound workloads.

Our **sequential approach** is a straight-forward execution structure in which a number of filters are processed sequentially in a pipeline, each frame is processed independently by one or more threads by dividing the frame spatially. In each pipeline stage, the worker threads are created, started, and eventually joined when finished. This would be the natural way of structuring the execution of a multithreaded media pipeline in a standalone application. Such processing pattern has natural barriers between each stage of the pipeline. For a execution system such as Cilk [10], Multicore Haskell [15], Threading Building Blocks [12] and others that use a work stealing model [2], the pattern of execution is similar to the sequential approach, but this depends very much on that way in which work units are assigned to worker threads, the workloads that are running simultaneously and scheduling order. Nevertheless, sequential execution is the baseline of our evaluation since it processes each filter in the pipeline in a *natural* order.

As an alternative execution structure, we propose using **backward dependencies** (BD) to execute workloads. This approach only considers the last stage of the pipeline for a spatial division among threads and such avoids the barriers between each pipeline stage. Furthermore, for each pixel in the output frame, the filter backtracks dependencies and acquires the necessary pixel(s) from the previous filters. This is done recursively and does not require intermediate pixels to be stored to memory. Figure 1 illustrates dependencies between three frames connected by two filters. The pixels in frame 2 are generated when needed by filter B using filter A. The BD approach has the advantage of only computing the pixels that are needed by subsequent filters. The drawback, however, is that intermediate data must be re-computed if they are

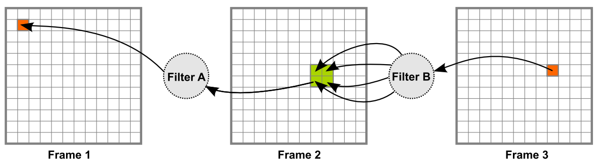


Figure 1. Backward dependencies (BD) example for two filters processing frames in a pipeline. The arrows indicate which pixels are required from the previous frame to generate the current pixel. Only one pixel's dependencies per frame are illustrated, other pixels have similar dependencies.

accessed multiple times because intermediate results are not stored. These re-computations can be mitigated by using the intermediate frames as buffer caches between filters, although the overhead of managing and checking this buffer cache can be large, which we see later in the paper.

The different approaches incur varying cache access patterns. Depending on memory access patterns, execution structure and chosen CPU microarchitecture, we expect the performance to change. The sequential approach accesses the buffers within a filter in sequential order, and the prefetch unit is thus able to predict the access pattern. A drawback of this approach is that data moved between filters do not necessarily reside in the cache of the core using the data last. First, this includes data whose cache line has been evicted and written back to a cache level with increased access time or memory. This may happen because the data size processed by a filter is larger than the amount of cache available, forcing write-back. Other reasons include context switches and shared caches. Second, output from a previous filter may not have been generated on the same core as the one that accesses the data, resulting in accesses to dirty cache lines on other cores. Given the spatial division of a frame within a filter, this sounds easy to avoid, but an area of input to a filter may result in output to a different spatial area, which the processor's prefetcher may not be able to predict. Thus, re-using the same core for the same part of a frame for multiple filters in a pipeline only increases cache locality for filters whose source pixels map spatially to the destination pixels.

To increase cache locality between filters, we also evaluate the BD approach, where data is accessed in the order needed to satisfy dependencies for the next filter in the pipeline. This ensures that pixels are accessed in a manner where data in between filters are likely to reside in the core's cache. That is to say, if the access pattern is not random, one can expect BD execution to always access spatially close memory addresses.

III. Experimental Setup

To evaluate the approaches and find which performs best in a low-level scheduler, we built an experimental framework supporting the proposed execution structures and wrote a set of image processing filters as a case study working on real-world data. The filters were arranged in different pipelines to induce behaviour differences that can impact performance.

All experiments measure exclusively computation time, i.e., the wall clock time of the parallel execution, excluding I/O and setup time. We use this instead of CPU time to further measurements on how good performance is actually possible,

Microarchitecture	CPU	Cores (SMT)	Private Cache	Shared Cache
Nehalem	Intel i5-750	4	64 kB L1 256 kB L2	8 MB L3
Sandy Bridge	Intel i7-2600	4 (8)	64 kB L1, 256 kB L2	8 MB L3
Sandy Bridge-E	Intel i7-3930K	6 (12)	64 kB L1, 256 kB L2	12 MB L3
Bulldozer	AMD FX 8192	8 (4x2)	64 kB L1[1], 2 MB L2[1]	8 MB L3

[1] Shared between two modules each having a separate integer unit while sharing an FPU.

Table I
MICROARCHITECTURES USED IN EXPERIMENTS.

Blur convolves the source frame with a Gaussian kernel to remove pixel noise.
Sobel X and Y are two filters that also convolve the input frame, but these filters apply the Sobel operator used in edge detection.
Sobel Magnitude calculates the approximate gradient magnitude using the results from Sobel X and Sobel Y.
Threshold unset every pixel value in a frame below or above a specified threshold.
Undistort removes barrel distortion in frames captured with wide-angle lenses. Uses bilinear interpolation to create a smooth end result.
Crop removes 20% of the source frame's height and width, e.g., a frame with a 1920x1080 resolution would be reduced to 1536x864.
Rotation rotates the source frame by a specified number of degrees. Bilinear interpolation is used to interpolate subpixel coordinates.
Discrete discretizes the source frame by reducing the number of color representations.
Binary creates a binary (two-colored) frame from the source. Every source pixel that is different from or above zero is set, and every source pixel that equals zero or less is unset.

Table II
IMAGE PROCESSING FILTERS USED.

Pipeline	Filter	Seq	BD	BD-CACHED
A	Blur	9.00	162.00	9.03
	Sobel X	9.00	9.00	9.00
	Sobel Y	9.00	9.00	9.00
	Sobel Magnitude	2.00	2.00	2.00
	Threshold	1.00	1.00	1.00
B	Undistort	4.00	10.24	2.57
	Rotate 6°	3.78	2.56	2.56
	Crop	1.00	1.00	1.00
C	Undistort	4.00	8.15	2.04
	Rotate 60°	2.59	2.04	2.04
	Crop	1.00	1.00	1.00
D	Discrete	1.00	1.00	1.00
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00
E	Threshold	1.00	3.19	0.80
	Binary	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
F	Threshold	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
	Binary	1.00	1.00	1.00
G	Rotate 30°	3.19	3.19	3.19
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00

Table III
EVALUATED PIPELINES AND THE AVERAGE NUMBER OF OPERATIONS PERFORMED PER PIXEL WITH DIFFERENT EXECUTION STRUCTURES.

since having used only half of the CPU time available does not mean that only half of the CPU's resources are utilized (cache, memory bandwidth, etc.). Also, by not counting I/O time, we remove a constant factor present in all execution structures, which we believe better captures the results of this study. Each experiment is run 30 times, and the reported computation time is the average All filters use 32-bit float computations, and overhead during execution, such as removing function calls in the inner-loop and redundant calculations, has been removed. Our data set for experiments consists of the two standard video test sequences *foreman* and *tractor* [18]. The former has a 352x288 pixel (CIF, 4:2:0) resolution with 300 frames of YUV data, the latter has 1920x1080 pixels (HD, 4:2:0) with 690 frames of YUV data.

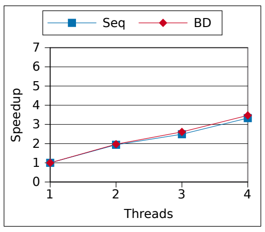
The experiments have been performed on a set of **modern microarchitectures** as listed in table I. The CPUs have 4 to 8 cores, and have rather different cache hierarchies: While the Nehalem has an L3 cache shared by all cores, operates at a different clock frequency than the cores and is called the *uncore*, the Sandy Bridge(-E) has a slice of the L3 cache assigned to each core and accesses the other parts using a ring interconnect running at core speed. Our specimen of the Bulldozer architecture consists of four modules, each of which containing two cores. On each module, L1 and L2 are shared between the two cores with separate integer units but a single shared FPU. We expected that these very different microarchitectures found and used in modern computing would produce very different program behaviour, and we have investigated how media workloads should be structured for execution on each of them to achieve the best performance.

We have developed a set of image processing **filters** for evaluating the execution structures. The filters are all data-intensive, but vary in terms of the number of input pixels needed to produce a single output pixel. The filters are later combined in various configurations referred to as pipelines. A short summary of the filters and their dependencies is given in table II.

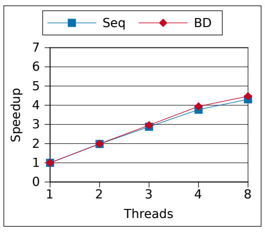
The filters are combined in various configurations into **pipelines** (as in figure 1). The tested pipelines are listed in table III. The pipelines combine the filters in manners that induce different amounts of work per pixel, as seen in the table. For some filters, not all intermediate data are used by later filters and are unnecessary to produce the final output. The BD approach will not produce these, e.g., a crop filter as seen in pipeline B will not require earlier filters to produce unused data. Another aspect that we expect to influence the results is cache prefetching. This means that by having filters that emit data in a different spatial position relative to its input, e.g. the rotation filter, we expect the prefetcher to contend fetching the relevant data.

IV. SCALABILITY

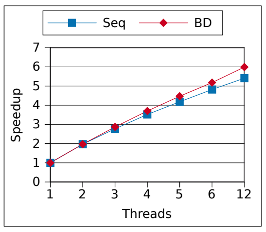
The pipelines are embarrassingly parallel, i.e., no locking is needed and they should therefore scale linearly with the number of cores used. For example, using four cores is expected to yield a 4x execution speedup. The threads created



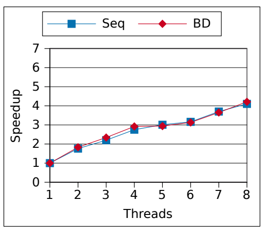
(a) Nehalem (4)



(b) Sandy Bridge (4)

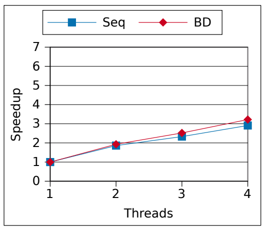


(c) Sandy Bridge-E (6)

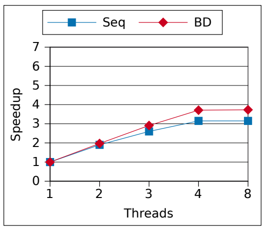


(d) Bulldozer (8)

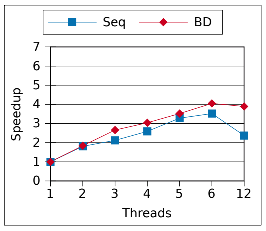
Figure 2. Individually normalized scalability of pipeline B running the tractor test sequence. Number of physical cores in parenthesis.



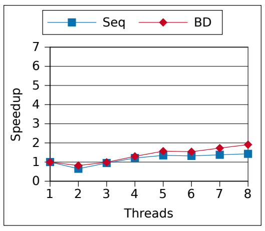
(a) Nehalem (4)



(b) Sandy Bridge (4)



(c) Sandy Bridge-E (6)



(d) Bulldozer (8)

Figure 3. Individually normalized scalability of pipeline B running foreman test sequence. Number of cores in parenthesis.

are handled by the Linux scheduler, which decides which thread to execute on which CPU (no affinity). Each new thread created works on its own frame-segment, but the last frame-segment is always processed by the thread that performs the I/O operations. In the single-threaded execution, both the I/O and processing operations are performed in the same thread, and therefore also on the same CPU core. Using two threads, we created one additional thread that is assigned the first half of a frame while the I/O thread processes the last half of the frame. We do this to minimize fetching source data from another core's private cache.

Figure 2 shows the relative speedup for the Nehalem, Bulldozer, Sandy Bridge and Sandy Bridge-Extreme microarchitectures that process pipeline B with the *tractor* sequence as input, comparing the sequential (Seq) and BD executions. Note that the running times of sequential and BD are individually normalized to their respective performance on one core, i.e., a higher value for BD does not necessarily imply better absolute performance than sequential. Looking at each architecture individually, we see that there is little difference in how the two execution methods scale on physical cores, but comparing architectures, we see that SB (figure 2(b)) is the only architecture that is able to scale pipeline B perfectly with the number of physical cores, as one could expect. SB-E (figure 2(c)) performs a little below perfect linear scaling achieved by its predecessor for this workload, and we see slightly better scalability results for BD execution than for sequential execution. On Nehalem (figure 2(a)), this pipeline doubles its performance with two threads, but after this, the increase in speedup per core diminishes. Bulldozer results (figure 2(d)) are even worse; from one to four threads we gain a 3x speedup, but there is not much to gain from using five to eight threads as we only manage to achieve a 4x speedup using the maximum number of cores. Bulldozer has four modules, each with two cores, but each module has only one FPU, and it is likely that this clamps performance to 4x. To summarize, the scalability results of pipeline B with the tractor sequence as input scales relatively well on Nehalem, Sandy Bridge and Sandy Bridge-Extreme using both execution modes. However, Bulldozer scales poorly as it only manages a 4x speedup using all of its eight cores.

Looking at scalability when testing pipeline B with the *foreman* sequence, the results become far more interesting as seen in figure 3. None of the four microarchitectures are able to achieve perfect scalability, and standing out is the Bulldozer plot in figure 3(d), where we see very little or no speedup at all using more threads. In fact, the performance worsens going from one to two threads. We look more closely into the behaviour of Bulldozer in section VII. Furthermore, we can see that the BD mode scales better than sequential for the other architectures. From one to two threads, we get close to twice the performance, but with more threads, the difference between sequential and BD increases.

To explain why pipeline B scales so much worse with foreman as input than tractor, one must take the differences into account. The foreman sequence's resolution is 352x288 (YUV, 4:2:0), which leads to frame sizes of 594 kB stored as floats. A large chunk of this can fit in private cache on the Intel architectures, and a full frame on Bulldozer. A frame in the tractor sequence has a resolution of 1920x1080 which requires almost 12 MB of data, exceeding even L3 size. In all pipeline stages except the last, data produced in one stage are referenced in the next, and if the source data does not reside in a core's cache, it leads to high inter-core traffic. Segmenting the frames per thread, as done when executing the pipelines in parallel, reduces the segments' sizes enough to fit into each core's private cache for the foreman frames, but not the larger tractor frames. Not being able to keep a large part of the frame in a core's private cache require continuous fetching from shared cache and/or memory. Although this takes time, it is the normal mode of operation. When an large part of a frame such as foreman fits in private cache after I/O, other cores that shall process this will have to either directly access the other core's private cache or request eviction to last-level cache on the other core, both resulting in high inter-core traffic. We were unable to find detailed information on this behaviour for the microarchitectures, but we can observe that scalability of this behaviour is much worse than that of the HD frame experiment. Moreover, since the BD mode only create one set of worker threads which are used throughout the lifetime of that pipeline cycle, and it does its computations in a recursive manner, the input data is likely to reside in the core's private cache. Also, since the BD mode does not require storing intermediate results and as such does not pollute the caches, we can see it scales better than sequential for the foreman tests.

With respect to the other pipelines (table III), we observed similar scalability across all architectures for both execution modes and inputs as seen in figure 2 and 3. In summary, we have shown that the BD mode provides slightly better scaling than sequential execution for our data-intensive pipelines on all architectures when the working unit to be processed (in this case a segment of a video frame) is small enough to a large extent reside in a core's private cache. Although scalability is beneficial, in the next section, we will look at the performance relative to sequential execution and see how BD and sequential perform against each other.

V. Performance

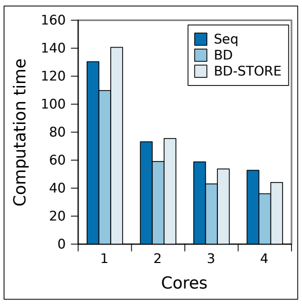
Our experiments have shown that achieving linear scaling on our data-bound filters is not trivial, and we have seen that it is especially hard for smaller units of work that mostly fit into a core's private cache and needs to be accessed on multiple cores. In addition, there are large microarchitectual differences visible. In this section, we look at the various pipelines as defined in section III to see how the sequential execution structure compares to backward dependency on various microarchitectures.

To compare sequential and BD execution, we have plotted computation time for pipeline A to G relative to their individual single-threaded sequential execution time using the foreman test sequence in figure 4. The plot shows that sequential execution provides best performance in most cases, but with some notable exceptions. The BD execution structure does not store and reuse intermediate pixels in any stage of the pipeline. Thus, when a pixel is accessed multiple times, it must be regenerated from the source. For example, if a filter in the last stage of a pipeline needs to generate the values of a pixel twice, every computation in every stage in the current pipeline involved in creating this output pixel must be executed twice. Obviously, this leads to a lot of extra computation as can be seen from table III, where for instance the source pixels are accessed 162 times per pixel for the BD approach in pipeline A, but only 9 times for the sequential approach. This is reflected in the much higher BD computation time than sequential for pipeline A for all plots in figure 4.

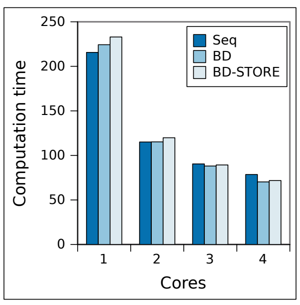
When looking at the other pipelines, we can see some very significant architectural differences. Again, Bulldozer stands out showing that for pipeline D, F, and G, the BD approach performs considerably better than sequential execution using all eight cores. Pipeline D and G perform better with BD execution, but it is rather unexpected for pipeline F, which does require re-computation of many intermediate source pixels. Still, the scalability achieved on Bulldozer for the foreman sequence were miniscule, as we saw in section IV.

The next observation we show is the performance of pipeline D. This pipeline performs the same amount of work regardless of execution structure, since every intermediate pixel is accessed only once. Most architectures show better performance for the BD approach, both for one and all cores. The only exception is the Sandy Bridge-E which performs slightly worse than sequential when using 6 cores. A similar behaviour as pipeline D is to be expected from pipeline G since it requires the same amount of work for both modes. This turns out not to be the case; for single-threaded execution on Nehalem and Bulldozer, pipeline G is faster using BD. Using all cores, also Sandy Bridge performs better using the BD approach. Sandy Bridge-E runs somewhat faster with sequential execution. We note that Sandy Bridge and Sandy Bridge-E are very similar architectures which ought to behave the same way. This turns out not to be the case, and even this small iteration of the microarchitecture may require different low-level schedules to get the highest level of performance - even for an embarrassingly parallel workload.

To find out if the performance gain seen with the BD approach is caused by better cache usage by keeping intermediate data in private caches or by the reduction of cache and memory pollution resulting from not storing intermediate results, we looked at pipeline D and G using BD while storing intermediate data. This mimics the sequential approach, although data is not referenced again later on, resulting in less cache pressure. Looking at figure 5, which shows pipeline D and G for Sandy Bridge, we can see that for a single core, the overhead of writing back intermediate results using BD-STORE results in worse performance than sequential execution, whereas this overhead diminishes when the number of threads is increased. Here, the BD-STORE structure outperforms sequential execution significantly. Accordingly, we ascribe the performance gain for the BD approach in this case to better private cache locality and usage than sequential execution.

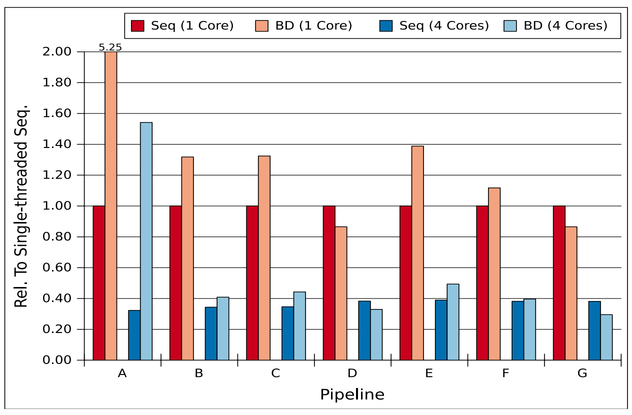


(a) Pipeline D

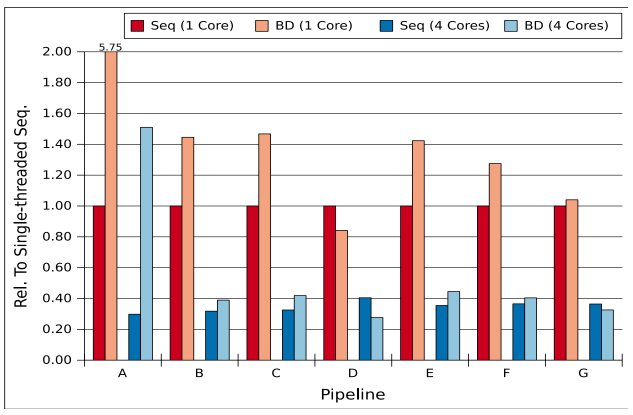


(b) Pipeline G

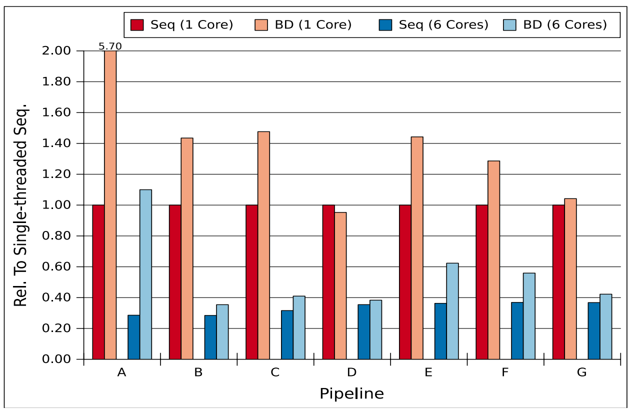
Figure 5. Computation time (ms) for foreman on Sandy Bridge.



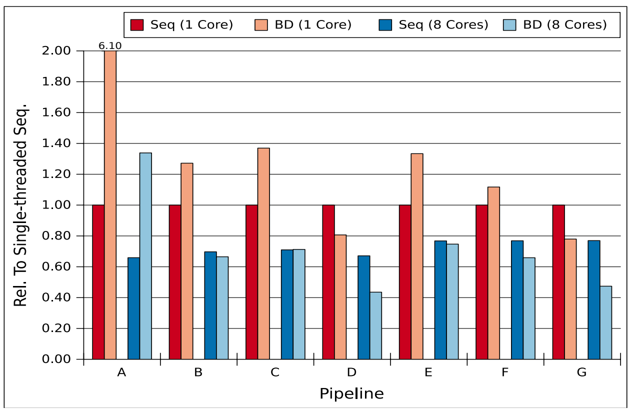
(a) Nehalem



(b) Sandy Bridge



(c) Sandy Bridge-E



(d) Bulldozer

Figure 4. Execution structure performance using the foreman dataset. Running times are relative to each pipeline's 1-core sequential time. Lower is better.

VI. Backward Dependency with Buffer Cache

Having shown that backward dependency execution structure performs better than sequential execution in some cases, we look at ways to improve the performance further. The main drawback of BD execution is, as we saw, that intermediate pixels must be recomputed when accessed multiple times, evident by pipeline A results. We have also shown that, when doing the same amount of work as the sequential approach, BD can perform better such as with pipeline D and G. In this section, we experiment with adding a buffer cache that keeps intermediate data for later reuse to mitigate this issue.

Instead of recursively generating all prior intermediate data when a pixel is accessed, the system checks a data structure to see if it has been accessed earlier. To do this without locking, we set a bit in a bitfield atomically to mark a valid entry in the buffer cache. It is worth noting that this bitfield consume a considerably amount cache space by using one bit for every pixel, e.g. about 37 kB for every stage in a pipeline for the foreman sequence. Other approaches for this buffer cache are possible, including a set of ranges and tree structures, but this is left for further work.

The cached results for pipelines A to G (BD-CACHED) using four threads on Sandy Bridge and processing foreman are shown in figure 6. We can see that the BD-CACHED approach provides better performance than sequential on pipeline B, D and G, but only B performs better than BD. Pipeline B has rotation and crop stages that reduce the amount of intermediate pixels that must be produced from the early filters compared to sequential execution (see table III). In comparison, pipeline C has also a significant reduction of intermediate pixel requirements, but we do not see a similar reduction in computation time. The only difference between pipeline B and C is the amount of rotation, with 6° for B and 60° for C. The skew from rotation for the pipelines causes long strides over cache lines when accessing neighbouring pixels from previous stages, which can be hard to prefetch efficiently. Also, parts of an image segment processed on a core may cross boundaries between segments, such that high inter-core traffic is required with sequential execution even though the same core processes the same segment for each stage. This is avoided with BD and BD-CACHED modes, but we can not predict which mode performs better in advance.

VII. Affinity

In section IV, we saw the diminishing performance when processing the foreman sequence with two and three threads on the Bulldozer architecture. When processing our pipelines in a single thread, we did not create a separate thread for processing, but processed in the thread that handled the I/O operations. Further, this lack of scaling was only observed

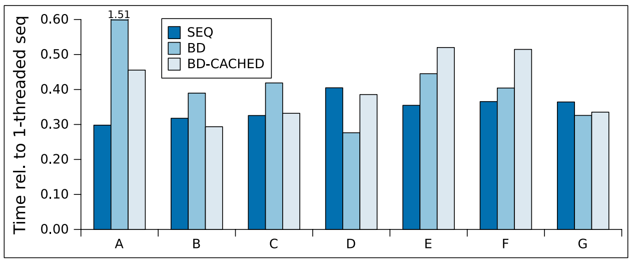


Figure 6. Computation time for Backward Dependency with buffer cache measured relative to single threaded sequential and tested with Sandy Bridge using 4 threads and foreman as input sequence.

when processing the low-resolution foreman sequence, for a large part of the frame could fit inside a core's private L2 cache. To investigate this further, we did an experiment where we manually specified each thread's affinity, which implies that we use separate threads doing I/O and processing, even in single-threaded execution.

Even though we only noticed the lack of scaling on the Bulldozer architecture, we tested the impact of affinity on the Nehalem and Sandy Bridge architectures as well (we omitted SB-E, which is very similar to Sandy Bridge). In figure 7, we have plotted the results for sequential processing of pipeline B while varying the number of threads. On Nehalem and Sandy Bridge, we tested two different execution modes, one mode where the processing of a frame was done on the same core that executed the I/O thread (IOSAME), while the second mode processed the frame on a different core than the I/O thread (IODIFF). Since the Bulldozer architecture has CPU modules, we added an additional mode for this architecture, in which processing threads were executed on different CPU modules than the thread handling I/O (MODULEDIFF).

We expected that processing on another core than I/O would increase the processing time, which was found to be only partially correct: From figure 7(a) we see that there are not any significant difference in processing on the same core as the I/O thread versus a different core on Sandy Bridge. Much of the same behaviour is seen on Nehalem using one thread, but when using two threads there is a large penalty of pinning these to different cores than the one doing I/O operations. In this case, using two threads and executing them on different cores than the I/O operations adds so much to the computation time that it is slower than the single threaded execution.

Looking at the Bulldozer architecture in figure 7(c), using only one thread we see that there are not any significant difference in which core or CPU module the processing thread is placed on. However as with Nehalem, when using two or more threads the IODIFF and MODULEDIFF execution modes have a huge negative impact on the performance. Even more unexpected, IOSAME and IODIFF using three threads are actually slower than IOSAME using two threads, and the fastest execution using eight threads completes in 1174 ms (omitted in plot), not much faster than what we can achieve with two threads.

In summary, we have seen that thread affinity has an enormous impact on Nehalem and Bulldozer when a large part of the data fits into the private cache. This can cause the two threads to have worse performance than a single thread when the data reside in another core's private cache. On Sandy Bridge, this limitation has been lifted and we do not see any difference due to affinity. Bulldozer is unable to scale to much more than two threads when the dataset fits into the private cache, presumably because the private cache is shared between two cores and the cost of doing inter-module cache access is too high.

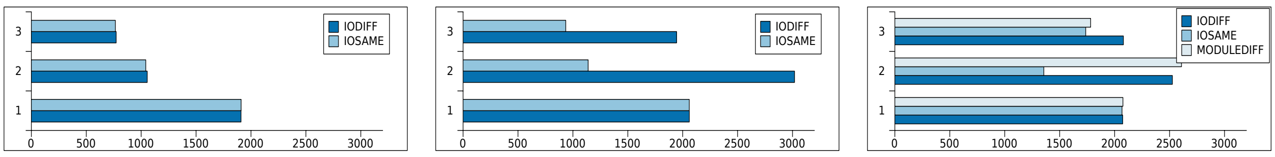
VIII. Discussion

The long-running data-intensive filters described in this paper deviate from typical workloads when looking at performance in terms of the relatively little computation required per unit of data. The filters used are primitive, but we are able to show large performance impacts by varying the order of execution and determining which core should do what. For such simple and embarrassingly parallel workloads, it is interesting to see the significant differences on how these filters perform on modern microarchitectures. As a case study, we chose image processing algorithms, which can intuitively be connected in pipelines. Real-world examples of such pipelines can be found in OpenCV [4], node-based software such as the Nuke [9] compositing tool and various VJ software. Other signal processing domains such as sound processing are applicable as well.

There is a big difference between batch processing and processing a stream of data, where in the former we can spread out independent jobs to a large number of cores, while in the latter we can only work on a limited set of data at a time. If we batch-processed the pipelines, we could instantiate multiple pipelines, each processing frames independently while avoiding the scalability issues that we experienced with foreman. In a streaming scenario, however, this is not possible.

The results shown in this paper are both unexpected and confusing. We had not anticipated such huge differences in how these modern processors perform with our workloads. With the standard parallel approach for such applications with either sequential execution of the pipeline or a work stealing approach, it is apparent that there is much performance to be gained by optimizing the order of operations for better cache usage. After all, we observe that the performance of the execution modes varies a lot with the behavior of a filter, e.g, amount of rotation applied by a filter. Moreover, it varies inconsistently with the number of threads, e.g., performance halved with two threads and sub-optimal processor affinity compared to a single thread. Thus, scheduling these optimally based on a-priori knowledge, we conjecture, is next to impossible, and profiling and feedback to the scheduler must be used to find the best configuration.

One option is to use profile-driven optimizations at compile time [5], where one or more configurations are evaluated and later used during runtime. This approach does, however, not work with dynamic content or tasks, i.e., if the parameters such as the rotation in pipeline B changes, the system must adapt to a new configuration. Further, the preferred configuration



(a) Sandy Bridge (b) Nehalem (c) Bulldozer

Figure 7. Execution times (ms) of pipeline B for the foreman video using different thread affinity strategies (lower is better).

may even change based on interactions between co-scheduled tasks, e.g., a CPU-bound and memory-bound task may perform better when co-scheduled than two memory bound tasks [8].

The preferred option then is to have a low-level scheduler that adapts dynamically to varying tasks, data, architectural limitations and shared resources. We propose an approach that uses instrumentation and allows a scheduler to gauge computation times of specific filters at execution time. Since the workloads we look at are periodic and long-running, the low-level scheduler can react to the observations and make appropriate adjustments, e.g., try different execution modes, data granularity, affinity and co-running competing workloads.

The prevalent approaches used in execution systems today are work stealing variants. Here, the system typically use a heuristics to increase cache locality such as spawning new tasks in the same queue; or stealing tasks from the back of another queue instead of the front to reduce cache contention, assuming tasks that are near in the queue are also near in terms of data. Although work stealing in its simplicity provides great flexibility, we have shown in this paper that execution order has a large performance impact on filter operations. For workloads that are long-running and periodic in nature, we can expect that an adaptive low-level scheduler will outperform the simple heuristics of a work stealing scheduler. An adaptive work stealing approach is feasible though, where work units are enqueued to the worker threads based on the preferred execution mode, data granularity and affinity, while still retaining the flexibility of the work stealing approach. Such a low-level scheduler is considered for further work. Another direction that we want to pursue is building synthetic benchmarks and looking at performance counters to pinpoint the cause of some of the effects that we are seeing, in particular with the Bulldozer microarchitecture.

IX. Conclusion

In this paper, we have looked at run-time considerations for executing (cyclic) streaming pipelines consisting of a data-intensive filters for media processing. A number of different filters and pipelines have been evaluated on a set of modern microarchitectures, and we have found several unexpected performance implications, within the well-known x86-family of microprocessors, like increased performance by backtracking pixel dependencies to increase cache locality for data sets that can fit in private cache; huge differences in performance when I/O is performed on one core and accessed on others; and different execution modes perform better depending on minor parameter variations in filters (or stages in the processing pipeline) such as the number of degrees to rotate an image. The implication of the very different behaviors observed on the different microarchitectures is a demand for scheduling that can adapt to varying conditions using instrumentation data collected at runtime. Our next step is therefore to design such a low-level scheduler in the context of our P2G processing framework [7].

Acknowledgments

We would like to thank Håkon Stensland for providing machines and insights to the differences observed. This work has been performed in the context of the *iAD* center for Research-based Innovation funded by the Norwegian Research Council.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of SPAA*, pages 1–12, 2000.
[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of SPAA*, pages 119–129, 1998.
[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of PPOPP*, pages 207–216, 1995.
[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
[5] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.
[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.
[7] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. P2G: A framework for distributed real-time processing of multimedia data. In *Proc. of ICPPW*, pages 416–426, 2011.
[8] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, Feb. 2010.
[9] T. Foundry. Nuke. http://www.thefoundry.co.uk/products/nuke/.
[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of PLDI*, pages 212–223, Montreal, Quebec, Canada, June 1998.
[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proc. of SIGMOD*, pages 1123–1134, 2008.
[12] Intel Corporation. Threading building blocks. http://www.threadingbuildingblocks.org.
[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, 2007.
[14] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. In *Proc. of Euro-Par 2008*, pages 151–161. 2008.
[15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009.
[16] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. of CC*, pages 179–196, 2002.
[17] Ž. Vrba, P. Halvorsen, and C. Griwodz. A simple improvement of the work-stealing scheduling algorithm. In *Proc. of MuCoCoS*, pages 925–930, 2010.
[18] XIPH. *XIPH.org Test Media*, 2012. http://media.xiph.org/video/derf/.
[19] YouTube. Holy nyans! 60 hours per minute and 4 billion views a day on youtube. http://youtube-global.blogspot.com/2012/01/holy-nyans-60-hours-per-minute-and-4.html, Jan. 2012.

Live demonstrations

Processing of Multimedia Data using the P2G Framework

Title: Processing of Multimedia Data using the P2G Framework [93].

Authors: P. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. B. Kristoffersen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 19th ACM international conference on Multimedia (MM), ACM, 2011.

Summary: A live demonstration of the P2G framework encoding video and dynamically adapting to the available resources was presented at ACM Multimedia.

A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS)

Title: A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS) [100].

Authors: K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Workshop on Network and Systems Support for Games (NetGames), IEEE / ACM, 2011.

Summary: A live demonstration of the LEARS lockless game server was presented at NetGames.

Transparent protocol translation and load balancing on a network processor in a media streaming scenario

Title: Transparent protocol translation and load balancing on a network processor in a media streaming scenario [42].

Authors: H. Espeland, C. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08.

Summary: A live demonstration of the proxy was presented at NOSSDAV.

Other research papers

The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - A flexible alternative to MapReduce

Title: The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - A flexible alternative to MapReduce [90].

Authors: Z. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen.

Published: The Journal of Supercomputing, Springer, 2010.

Abstract: Even though shared-memory concurrency is a paradigm frequently used for developing parallel applications on small- and middle-sized machines, experience has shown that it is hard to use. This is largely caused by synchronization primitives that are low-level, inherently nondeterministic, and, consequently, non-intuitive to use. In this paper, we present the Nornir run-time system. Nornir is comparable to well-known frameworks such as MapReduce and Dryad that are recognized for their efficiency and simplicity. Unlike these frameworks, Nornir also supports process structures containing branches and cycles. Nornir is based on the formalism of Kahn process networks, which is a shared-nothing, message-passing model of concurrency. We deem this model a simple and deterministic alternative to shared-memory concurrency. Experiments with real and synthetic benchmarks on up to 8 CPUs show that per- formance in most cases scales almost linearly with the number of CPUs, when not limited by data dependencies. We also show that the modeling flexibility allows Nornir to outperform its MapReduce counter-parts using well-known benchmarks.

Limits of work-stealing scheduling

Title: Limits of work-stealing scheduling [94].

Authors: Z. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz.

Published: Job Scheduling Strategies for Parallel Processing (14th International Workshop), Springer, 2009.

Abstract: The number of applications with many parallel cooperating processes is steadily increasing, and developing efficient runtimes for their execution is an important task. Several frameworks have been developed, such as MapReduce and Dryad, but developing scheduling mechanisms that take into account processing *and* communication requirements is hard. In this paper, we explore the limits of work stealing scheduler, which has empirically been shown to perform well, and evaluate load-balancing based on graph partitioning as an orthogonal approach. All the algorithms are implemented in our Nornir runtime system, and our experiments on a multi-core workstation machine show that the main cause of performance degradation of work stealing is when very little processing time, which we quantify exactly, is performed per message. This is the type of workload where graph partitioning has the potential to achieve better performance than work-stealing.

Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation

Title: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation [95].

Authors: H. Espeland, H. K. Stensland, D. H. Finstad, and P. Halvorsen

Published: International Journal of Multimedia Data Engineering and Management (IJMDEM) 3(2):1-19, IGI Global, 2012.

Abstract: Segmented adaptive HTTP streaming has become the de facto standard for video delivery over the Internet for its ability to scale video quality to the available network resources. Here, each video is encoded in multiple qualities, i.e., running the expensive encoding process for each quality layer. However, these operations consume both a lot of time and resources, and in this paper, the authors propose a system for reusing redundant steps in a video encoder to improve the multi-layer encoding pipeline. The idea is to have multiple outputs for each of the target bitrates and qualities where the intermediate processing steps share and reuse the computational heavy analysis. A prototype has been implemented using the VP8 reference encoder, and their experimental results show that for both low- and high-resolution videos the proposed method can significantly reduce the processing demands and time when encoding the different quality layers.

LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Title: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition [96].

Authors: K. Raaen, H. Espeland, H. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2012 International Conference on Parallel Processing Workshops, IEEE, 2012.

Abstract: Supporting thousands of interacting players in a virtual world poses huge challenges with respect to processing. Existing work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players needs to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency or inconsistency. It is a major challenge of game engines to scale such areas to the largest number of players possible; in a deviation from earlier thinking, parallelism on multi-core architectures is applied to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that need timely updates. We also measure how the response time for the multi-threaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game-server can be an embarrassingly parallel problem.

Improved Multi-Rate Video Encoding

Title: Improved Multi-Rate Video Encoding [97].

Authors: D. H. Finstad, H. K. Stensland, H. Espeland, and P. Halvorsen

Published: Proceedings of the International Symposium on Multimedia (ISM), IEEE, 2011.

Abstract: Adaptive HTTP streaming is frequently used for both live and on-Demand video delivery over the Internet. Adaptiveness is often achieved by encoding the video stream in multiple qualities (and thus bitrates), and then transparently switching between the qualities according to the bandwidth fluctuations and the amount of resources available for decoding the video content on the end device. For this kind of video delivery over the Internet, H.264 is currently the most used codec, but VP8 is an emerging open-source codec expected to compete with H.264 in the streaming scenario. The challenge is that, when encoding video for adaptive video streaming, both VP8 and H.264 run once for each quality layer, i.e., consuming both time and resources, especially important in a live video delivery scenario. In this paper, we address the resource consumption issues by proposing a method for reusing redundant steps in a video encoder, emitting multiple outputs with varying bitrates and qualities. It shares and reuses the computational heavy analysis step, notably macro-block mode decision, intra prediction and inter prediction between the instances, and outputs video in several rates. The method has been implemented in the VP8 reference encoder, and experimental results show that we can encode the different quality layers at the same rates and qualities compared to the VP8 reference encoder, while reducing the encoding time significantly.

Improving File Tree Traversal Performance by Scheduling I/O Operations in User space

Title: Improving File Tree Traversal Performance by Scheduling I/O Operations in User space [98].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen.

Published: Proceedings of the 28th IEEE International Performance Computing and Communications Conference (IPCCC), IEEE, 2009.

Abstract: Current in-kernel disk schedulers provide efficient means to optimize the order (and minimize disk seeks) of issued, in-queue I/O requests. However, they fail to optimize sequential multi-file operations, like traversing a large file tree, because only requests from one file are available in the scheduling queue at a time. We have therefore investigated a user-level, I/O request sorting approach to reduce inter-file disk arm movements. This is achieved by allowing applications to utilize the placement of inodes and disk blocks to make a one sweep schedule for all file I/Os requested by a process, i.e., data placement information is read first before issuing the low-level I/O requests to the storage system. Our experiments with a modified version of tar show reduced disk arm movements and large performance improvements.

Improving Disk I/O Performance on Linux

Title: Improving Disk I/O Performance on Linux [99].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, A. Petlund, and P. Halvorsen.

Published: UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference, GUUG, 2009.

Abstract: The existing Linux disk schedulers are in general efficient, but we have identified two scenarios where we have observed a non-optimal behavior. The first is when an application requires a fixed bandwidth, and the second is when an operation performs a file tree traversal. In this paper, we address both these scenarios and propose solutions that both increase performance.