universität wien

Masterarbeit

Titel der Masterarbeit

"Programming Support for High-Level Parallel Pipeline Patterns on Hybrid Many-core Architectures"

verfasst von

Enes Bajrovic, BSc.

angestrebter akademischer Grad

Diplom-Ingenieur (DI)

Wien, 2014

Studienkennzahl lt. Studienblatt: A 066 940
Studienrichtung lt. Studienblatt: Scientific Computing
Betreut von: Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner

Dedicated to my parents, sister and brother

Abstract

The increasing heterogeneity in modern hardware architectures promises significant performance gains as well as a better performance-to-power ratio. New hardware appears to be leaning towards hybrid many-core architectures that combine conventional multi-core systems with accelerators, such as GPUs , Intel Xeon Phis, etc. However, developing software that can fully exploit the potential computing power proves to be considerably complex and the programmability of such systems is significantly hindered. Consequently, programmers are forced to explicitly deal with distinct memory spaces, locality, load balancing, while using different programming models and parallelization strategies. In order to enable efficient execution and increase programmability, many approaches combine higher-level abstractions in programming models with compilers and runtimes systems.

This thesis presents an approach towards the support for high-level pipeline parallel patterns on heterogeneous many-core architectures. It builds on top of a component-based, task-parallel programming model developed within the European project PEPPHER. In this context tasks relate to the components that may have multiple implementation variants tailored for different execution units of hybrid many-core system. A sophisticated heterogeneous runtime system is responsible for mapping suitable component implementation variants and scheduling their execution to different execution units. On top of these foundations, we have developed a framework comprised of a high-level coordination language, a source-to-source compiler and the coordination layer for pipelining that relies on a heterogeneous runtime system to perform dynamic scheduling in a performance and resource efficient way. We evaluate the approach on two different heterogeneous systems and demonstrate its viability and effectiveness with two benchmarks from the areas of computer vision and bioinformatics.

Zusammenfassung

Heterogene Elemente in modernen Hardware-Architekturen versprechen umfassende Leistungssteigerungen sowie verbesserte Energieeffizienz. Neue Hardware Entwicklungen scheinen diesen Trend durch die Kombination von konventionellen multi-core Prozessoren mit Hardware-Beschleunigern wie etwa GPUs oder Intel Xeon Phi zu übernehmen. Die Softwareentwicklung für derartige Systeme gestaltet sich jedoch äußerst komplex und es ist oftmals schwierig, deren gesamtes Leistungspotenzial zu nutzen. ProgrammiererInnen sind gezwungen explizit auf verschiedene Speicherbereiche, Lokalität und Lastverteilung Rücksicht zu nehmen und müssen darüber hinaus verschiedene Programmiermodelle und Parallelisierungsverfahren beachten. Um hohe Effizienz sowie Programmierbarkeit zu gewährleisten, kombinieren bestehende Ansätze oftmals höhere und abstrakte Programmiermodelle mit Compiler-Techniken sowie Laufzeitsystemen.

Diese Arbeit präsentiert einen Ansatz für die Programmierung von heterogenen many-core Systemen mit Hilfe eines parallelen Pipeline Programmiermuster. Dies baut auf einem komponentenbasierten, task-parallelen Programmiermodell welches innerhalb des europäischen Forschungsprojekt PEPPHER entwickelt wurde. Das gewählte Komponentenmodell ermöglicht verschiedene Implementierungsvarianten von Task-Routinen welche für verschiedene Zielarchitekturen angepasst werden können. Ein hochentwickeltes Laufzeit-System ist für die Verteilung und Auswahl passender Implementierungsvarianten sowie deren effizienter Ausführung auf verschiedenen Hardware-Elementen verantwortlich. Aufbauend auf dieser Basis, haben wir ein Programmiersystem entwickelt welches aus einer abstrakten Koordinationssprache, einem Source-to-Source Transformationssystem sowie einer Laufzeit-Koordinationsbibliothek für Pipelining besteht. Der vorgestellte Ansatz wird auf zwei unterschiedlichen heterogenen Systemen mit Anwendungen aus Computer-Vision sowie Bioinformatik evaluiert.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1

1.1 Motivation . . . 2
1.2 Programming Parallel Systems . . . 6
1.3 Goal of the Thesis . . . 8
1.4 Used Technologies . . . 8
1.5 Developed Software . . . 10
1.6 Structure of the Thesis . . . 11

2 PEPPHER Approach 13

2.1 Introduction . . . 13
2.2 Component Model . . . 14
2.3 Execution Model . . . 20
2.4 Coordination Primitives . . . 22

3 Pipeline Pattern 27

3.1 Introduction . . . 27
3.2 Performance Considerations . . . 31
3.3 Pipeline Directives . . . 32

4 Coordination Layer 37

4.1 Overview . . . 38
4.2 Structure . . . 40
4.3 Pipelines . . . 41
4.4 Stages . . . 43
4.5 Buffer Mechanisms . . . 47

4.6 Pipeline Manager 50
4.7 Pipeline Execution and Coordination 54
4.8 Runtime and Scheduling 57
4.9 Other classes 59
4.10 Performance Measurement Facilities 60

5 Source-To-Source Transformation **65**
5.1 Rose Framework 65
5.2 Transformation Process 67
5.3 Transformation Example 69

6 Experiments and Evaluation **75**
6.1 Target Hardware 76
6.2 Face Detection Application 77
6.3 Needleman-Wunsch Application 85
6.4 Summary 91

7 Related Work **93**

8 Conclusion and Future Work **95**

List of Figures **99**

List of Tables **101**

Abbreviations **103**

Bibliography **105**

1 Introduction

There ain't no such thing as a free lunch.

(from "The Moon Is a Harsh Mistress", R. A. Heinlein)

The quotation refers to an article written by Herb Sutter in 2005 [1], in which he addresses the shift toward concurrency in software caused by ongoing changes in parallel hardware architectures over the last decade. Furthermore, ever-increasing parallelism in hardware is accompanied by heterogeneity commonly achieved by coupling different special-purpose cores or accelerators in combination with homogeneous multi-core. Although this approach delivers means to quantitatively increase computational power and improve performance-to-power ratio in hardware, it is notoriously complex to deal from a software perspective. In addition to concurrency, programmers need to deal with complexity by utilizing different programming models and parallelization strategies to address full potential of the system and achieve performance gains.

Consequently, simplifying parallel programming has become a target of many research efforts, exploring options in relevant areas of computer science - from compilers and programming languages to programming patterns and concrete algorithms. In this work, we explore how parallel programming patterns [2] expressed at high-level of abstraction can be used in combination with compiler and runtime system to fully utilize heterogeneous systems. In the context of the European project "PEPPHER", we have developed a framework for high-level parallel pipeline patterns comprised of a coordination language, a source-to-source compiler and an object-oriented coordination layer for pipelining, targeting single-node heterogeneous many-core systems.

This chapter briefly covers recent hardware and software trends that motivated this work and presents the goals we aim to achieve. Additionally, we provide a summary of used technologies and present the structure of the thesis.

1.1 Motivation

1.1.1 Hardware Trends - A Brief Overview

In 1965th co-founder of Intel Gordon E. Moore, after empirical observations, predicted that the number of transistors on a chip, for minimum components cost, will roughly double every eighteen months to two years [3, 4, 5]. More transistors mean more instructions per second, which directly influences an increase in processing power. Later, his statement became known as “Moore’s Law” and served its purpose as a goal to achieve in the entire IT industry. The law has been sustained so far and it is predicted that it will be sustained for another several chip generations. However, recent trends in hardware point to the conclusion that the way how computing power is exploited is changing. In 2005 Moore stated:

”It can’t continue forever. The nature of exponentials is that you push them out and eventually disaster happens. In terms of size [of transistor] you can see that we’re approaching the size of atoms which is a fundamental barrier, but it’ll be two or three generations before we get that far - but that’s as far out as we’ve ever been able to see. We have another 10 to 20 years before we reach a fundamental limit. By then they’ll be able to make bigger chips and have transistor budgets in the billions.” [5]

Sustainability of Moore’s Law depends on the ability to technologically facilitate an increase in the quantity of instructions executed per time unit, which implies an increase in computing capacity. However, as higher clock rates correspond to an exponential increases in temperature, it recently become obvious that it is more and more difficult to manage chip power and heat caused by limitations in the design and architecture of CPUs developed up to now [6]. This led to the conclusion that increasing the clock frequency might not be the only method of increasing computation speed and keep sustained growth. In the literature, these issues are often described as the *power wall* (or the *frequency wall*) [7]. As the matter of fact, the same technical report presents not one, but three walls defining the end of times of automatic increase in computing performance with the equation: *Power Wall +Memory Wall + ILP Wall = Brick Wall* [7].

One approach to solve this problem is the introduction of multiprocessor configurations in hardware systems by placing multiple cores on a single processor chip. In other words, parallelism is used rather than frequency scaling (see Figure 1.1). The apparent shift is visible

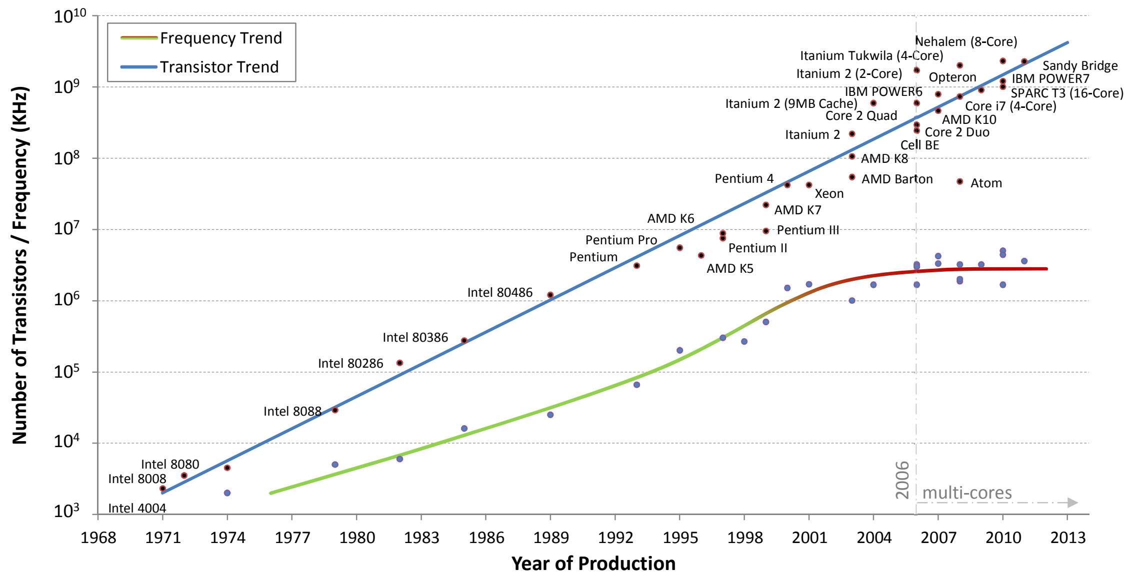


Figure 1.1: Moore's Law still holds, however frequency scaling era appears to be over.

in "Top500" lists. Since 1993, twice a year, the "Top500" project benchmarks and ranks the five hundred most powerful computers in the world. It provides an interesting overview of how parallel hardware systems dynamically evolved in the last few decades, experiencing many changes in their architectures. Not only there is a notable shift from single to multi-core chips, but also there is a notable diversity among parallel architectures.

According to the recently released lists, current top systems are petaflop-class systems capable of performing quadrillions of floating point operations per second such as Cray's"Titan", IBM's "Sequoia", Fujutsu's "K computer", NUDT's 'Tianhe-1A" and "Tianhe-2 (MilkyWay-2)", Cray's "Jaguar", "Nebulae" by Dawning, IBM's "Roadrunner", etc. [8]. These top systems are representatives of architecturally different approaches of building supercomputers with parallelism present at different levels.

All of the current top supercomputers are comprised of many clustered compute nodes. On the one hand, there are systems utilizing nodes comprised of the homogeneous multi-core CPUs such as "K computer" employing SPARC64 VIIIfx 2.0GHz CPUs with custom interconnect reaching performance of around 10.5 PFlops or Cray's "Jaguar" based on Cray XT5 and XT4 models, and connects 224256 2.6GHz cores in total, delivering performance of 1.759 PFlops. On the other hand, usually less power hungry, but significantly more complex to program and use are the systems of clustered nodes comprising non-homogeneous processing units and often relying on accelerators such as Graphics Processing Units (GPUs), Cell Broadband Engine (Cell BE) or Xeon Phi processors to provide computing speed. IBM's "Roadrun-

ner" is one of the first heterogeneous (or hybrid) supercomputers, combining heterogeneous Cell BE processors and x86 processors. The whole system connects 3060 "tri-blade" nodes, each comprised of two IBM QS22 blade servers and one IBM LS32 blade server, hence the name. The roadrunner's performance reaches petaflop scale, with the maximum observed performance of 1.042 PFlops.

The current top system is NUDT's "Tianhe-2 (MilkyWay-2)" housed at the National Super Computer Center in Guangzhou, China. The system combines Intel Xeon E5-2692 12C 2.200GHz with Intel Xeon Phi 31S1P units, reaching 33.8 PFlops, with theoretical peak of almost 55 PFlops. Other heterogeneous representatives are "Tianhe-1A" and "Nebulae", which are comprised of regular CPUs and NVIDIA or AMD GPUs reaching performance of ca. 2.5 PFlops and 4.7 PFlops. Cray's "Titan" - Cray XK7 comprises 18688 16-core AMD Opteron CPUs and the same number of NVIDIA Tesla K20 GPUs and reaches 17.590 PFlop/s of the observed performance whilst consuming 8209.00 kW.

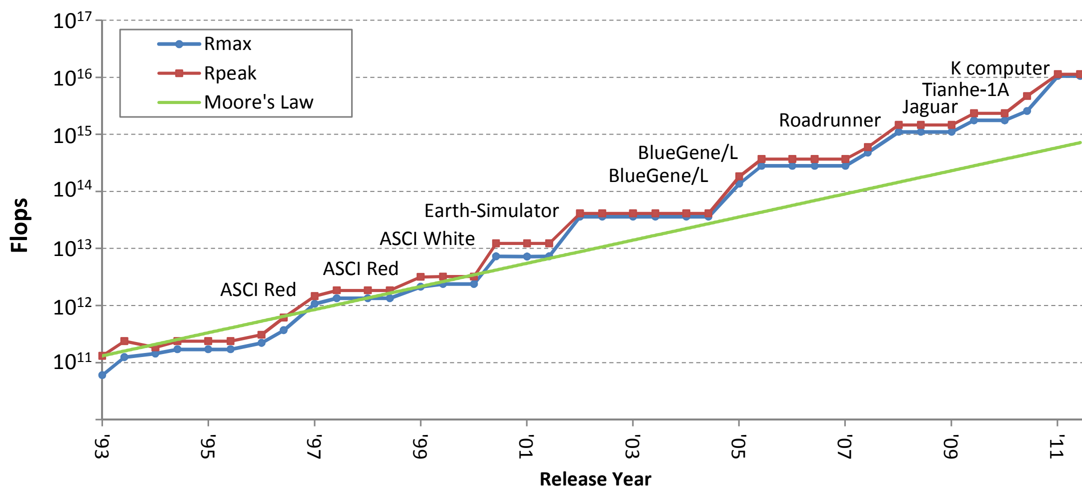


Figure 1.2: Peak (Rpeak) and achieved (Rmax) performance of the top supercomputers since the measurements started in 1993.

The evident prevalence of multi-core systems with or without accelerators exceeds the boundaries of large-scale systems, thus making parallel computing essential for all areas of IT, not just High Performance Computing (HPC). The full performance potential of multi-core systems can only be exploited through parallelism i.e. applications have to be parallelized. As hardware complexity increased over time, productivity of programmers decreased, making development and porting of software more and more expensive.

1.1.2 Homogenous and Heterogeneous Multi-Core/Many-Core Systems

It is not uncommon for modern single-node multi-core systems to comprise several different processing units. Such systems are usually referred to as *heterogeneous* multi-cores. Contrary, multi-core systems comprising only cores of the same design are called *homogeneous* multi-cores. These architectures usually employ a shared memory model, where all major memory resources are mapped on one address space and are accessible by all processor cores and/or direct memory access (DMA) controllers. However, most of the current processors use caches on several levels (e.g: L1, L2 and L3 caches) to improve performance by exploiting locality of reference in memory accesses. As the number of cores in a chip increases, the overhead introduced by keeping data coherent becomes significant. In traditional processors this is ensured by the cache coherency protocol, however "instruction-level parallelism (ILP) wall" issues [9] limit the amount of ILP in the future multi-cores.

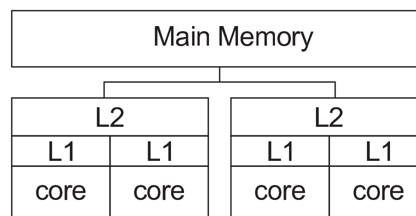


Figure 1.3: High-level representation of homogeneous multi-core CPU. The figure depicts quad-core architecture with L1 and L2 caches. L2 caches are shared among two cores.

A shared memory system's single address space and homogeneity of processing cores makes programming the homogeneous multi-cores significantly easier than on the heterogeneous ones. One example is Open Multi-Processing (OpenMP) [10], which uses compiler directives to enable parallelism with well-known programming languages (with C, C++ and Fortran). OpenMP uses shared memory model and fork-join model of parallel execution. However, lack of the notion of non-uniform memory access model, makes it less attractive for the heterogeneous systems [2].

The heterogeneous multi-cores often comprise processing units with their own local memory address spaces. Although the idea to increase computing capacity of a system by attaching additional computing elements is straight forward and effective in terms of raw compute power, it usually brings many programming challenges. Such systems often comprise differ-

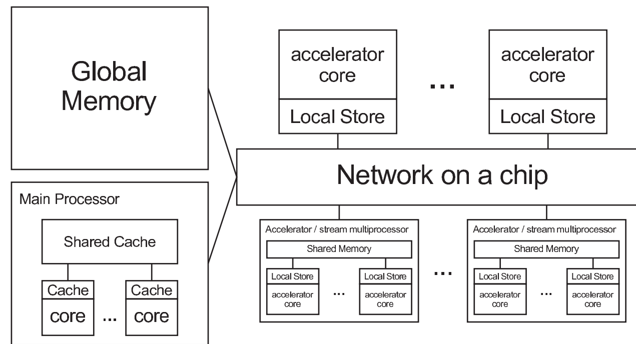


Figure 1.4: High-level representation of heterogeneous multi-core CPU.

ent instruction set architectures, distinct memory spaces that are farther away from the main memory, thus rendering data transfers more costly. Since none of the issues are handled in hardware, the job of dealing with the distributed memory model, where each compute node has its own private memory, orchestration of the data transfers, synchronization and load balancing is shifted to software.

Moreover, there is a clear trend towards increasing the number of different execution units in the system. On the one side, multi-cores are evolving into many-cores often packed with special-purpose and power-efficient cores (e.g., Cell BE [11], AMD Accelerated Processing Unit (APU) [12]). On the other hand, major source of power-efficient computating comes in for of the accelerators such as GPUs or Xeon Phis [13], that can be easily attached to system.

1.2 Programming Parallel Systems

From the software perspective, exploiting parallelism is guided by the fact that any program can usually be decomposed into smaller, discrete computational parts and (with some coordination) executed on multiple execution units [14]. Essentially, there are two key concepts in this context that address exactly this decomposition - data and task parallelism, both being highly relevant to today's mainstream hardware architectures.

With *data parallelism* (or data-based decomposition [14]) the idea is to partition data and achieve parallelism by applying the same computation on different parts of the dataset. Assuming there are no data dependencies, the computations operating on different parts of the dataset may execute in parallel. A typical, although very simple example of data parallel execution is addition of two vectors, where each addition operation may be performed in

parallel. Accelerators, such as GPUs, make heavy use of data parallelism. For example, in CUDA architecture, threads are scheduled in a groups called *warps*. For each warp one instruction is fetched and executed for all threads in that warp, following the single instruction, multiple data (SIMD) model.

Task parallelism is related to the functional (or computation) decomposition of the software. In this approach a program is statically or dynamically divided into separate units of computational work (*tasks*), that can be mapped to different execution units and eventually executed in parallel. In this case the decomposition may be represented by a directed acyclic graph (DAG) with nodes representing the tasks and edges relating to the task dependencies. This concept is pushed one step further in task-based programming models where task decomposition is combined with a runtime system in order to better exploit available parallelism.

In general, implementing parallelism in software can be a very complex task. To use the potential computing power efficiently, programmers often need to determine how to parallelize code for the system. This implies they have to perform data and/or functional decomposition, identify the portions of the code that would likely be beneficial to execute on accelerators, orchestrate the data movement between global memory and local memory stores of the accelerators and coordinate execution in order to utilize all of the computing abilities of hardware. This can lead to a very long and tedious process and the eventual success does not imply satisfactory performance and the portability of such application. Therefore, new programming models aim to support high productivity and high performance and yet be widely usable.

In the scientific computing domain some responses to these issues are reflected in the design of several programming languages like High Performance Fortran (HPF) [15], Unified Parallel C (UPC) [16], Co-Array Fortran [17] and Titanium [18]. However, despite its productivity limitations ([19], [20], [2]) Message Passing Interface (MPI) represents the standard and most frequently used model to achieve high performance on large-scale distributed systems. On the other hand, a lot of research effort is invested into the area of supporting and/or extending current programming models, runtime support and compilers for the new multi-core architectures (such as Cilk [21], HMPP [22], StarPU [23], StreamIt [24], etc.). Such approaches aim to ease the task of parallel programming, expose parallelism to a programmer and/or rise the level of abstraction programmer needs to deal with. Moreover, high-level parallel patterns found in software are utilized to simplify parallel programming [14, 2]. Often, the support for such patterns is combined with runtime systems and/or compilers with a goal to improve various aspects of the software such as programmability, performance and portability.

1.3 Goal of the Thesis

The primary goal of this thesis is the development of a framework for supporting tuneable high-level parallel pipeline patterns on heterogeneous many-core architectures. The pipeline patterns are particularly useful when full parallelization of a loop is not possible due to data dependencies and when significant amount of data needs to be processed. Additionally, we provide a mechanism for users to explicitly influence and tune such parallel patterns via source code annotations. The whole system comprises three major parts: (1) a high-level coordination language for expressing parallel pipeline patterns, (2) an object-oriented coordination layer for pipelining that utilizes a heterogeneous runtime system in order to make use of task, data, and pipeline parallelism potentially exploitable on a particular hardware, and (3) a source-to-source compilation system that transforms user-annotated code containing high-level pipeline patterns to a code that utilizes the pipeline coordination layer.

1.4 Used Technologies

Table 1.1 summarizes all hardware and software technologies used in this work.

Technology	Description
C/C++	General purpose programming languages, The GNU Compoler Collection. http://gcc.gnu.org
CUDA	Compute Unified Device Architecture (CUDA) - parallel computing platform and programming model created by NVIDIA, http://nvidia.com/cuda
OpenCL	Open Computing Language (OpenCL) standard for parallel programming of heterogeneous architectures, https://www.khronos.org/opencl
StarPU	A unified runtime system that offers support for heterogeneous multi-core architectures (General-purpose Computing on Graphics Processing Units (GPGPU), IBM Cell BE, ...). https://gforge.inria.fr/projects/starpu

Rose Compiler Framework	An open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C(C89 and C98), C++(C++98 and C++11), UPC, Fortran (77/95/2003), OpenMP, Java, Python and PHP applications. http://www.rosecompiler.org
RedHat Enterprise Linux 5.5	Commercial linux-based operating system developed by Red Hat
Intel TBB Library	C++ template library for task parallelism. http://threadingbuildingblocks.org
Boost Libraries	Portable C++ source libraries. http://www.boost.org
OpenCV	Open Source Computer Vision (OpenCV), http://opencv.willowgarage.com
Rodinia Benchmar Suite	Rodinia Benchmark Suite, http://lava.cs.virginia.edu/Rodinia/
PHIA GPU cluster	Eight node hybrid computational cluster. Each node comprises two Intel Xeon octa-core E5-2650 2.60Ghz (Sandy Bridge) CPUs , 128GB system memory and four accelerator cards featuring different combinations of Nvidia Tesla K20 M-class cards and Xeon Phi 5110P cards.
NADIA system	The system comprises two Intel Xeon octa-core E5-2650 2.60Ghz (Sandy Bridge) CPU and 128GB system memory, as well as three GPUs: Nvida GTX 480 GPUs, Nvida GTX 285 and ATI Radeon HD5870.
CORA GPU cluster	Eight node hybrid computational cluster. Each node comprises two Intel Xeon quad-core X5550 2.66Ghz CPU and 24GB DDR3-1333 system memory and hosts two nVidia Tesla C2050 GPUs and one nVidia Tesla C1060 GPU.

Table 1.1: Software and hardware technologies used in the thesis.

1.5 Developed Software

Table 1.2 summarizes all software that has been developed in this thesis.

Technology	Description
High-Level Coordination Language	A high-level coordination language support for PEPPHER parallel pipeline patterns in the form of C/C++ preprocessor directives.
Coordination Layer for pipelining	A C++ runtime library that supports multi-threaded parallel execution of the pipelines on heterogeneous many-core architectures with the goal of utilizing all available execution units simultaneously. The coordination layer has been written in roughly 4250 lines of code.
Transformation Tool	A source-to-source compiler supporting automatic transformation of codes with PEPPHER high-level parallel pipeline patterns to a code that utilizes the coordination layer for pipelining. The transformation tool has been realised with roughly 2500 lines of source code.
Face Detection Application	An image processing pipeline for detecting human faces on a stream of images. All application versions have total of around 900 lines of code.
Needleman-Wunsch Application	An application that performs multiple pairwise alignment of many DNA sequence pairs using Needleman-Wunsch algorithm. All application versions required roughly 1400 lines of code.

Table 1.2: Software developed in the thesis.

1.6 Structure of the Thesis

The development of our prototype framework is accomplished within the context of the PEPPHER project. In Chapter 2 we describe the PEPPHER approach, while focusing mostly on the PEPPHER component model. Chapter 3 introduces the basic ideas behind pipelining in the context of the multi-core processors, as well as the support for high-level coordination constructs. Afterwards, we describe the object-oriented coordination layer for pipelining in Chapter 4, and the transformation process in Chapter 5. We present the evaluation results in Chapter 6, elaborate on the related work in Chapter 7, and finally conclude with Chapter 8.

2 PEPPHER Approach

2.1 Introduction

PEPPHER (Performance Portability and Programmability for Heterogeneous Many-core Architectures) was a three-year European FP7 project that addressed programmability and performance portability for single-node heterogeneous many-core systems [25].

PEPPHER utilizes an extended component-based approach, where performance-critical parts of the application can be represented as multi-architectural components. Such components may have multiple (possibly parallelized) implementation variants tailored for different hardware configurations (e.g. different number or types of execution units, memory layouts, interconnect, etc). In addition, PEPPHER components are enriched with meta-data and made *performance-aware* by associating the analytical or history based performance models. A combined approach of static composition techniques for pre-selection and dynamic *resource-aware* runtime scheduling mechanisms allows suitable *performance-aware* components to be efficiently scheduled on available system resources. The performance information, available system resources, input data layout and data availability are taken into account in order to optimize execution time, power consumption or other relevant criteria [26].

The holistic approach taken by PEPPHER, builds on well-established software technologies and relies on existing parallel programming models and languages for internal parallelization of the components. It aims to develop a methodology and guidelines for constructing portable and performant software for hybrid hardware architectures [27]. This chapter gives only a brief introduction to the PEPPHER framework. More information can be found in the PEPPHER Project deliverables D1.1 [25], D1.2 [28], D1.4 [29].

The key parts of the PEPPHER framework for programming and optimizing applications for heterogeneous architectures relevant to this work are outlined in the following list:

- A flexible and extensible component model for encapsulating and annotating performance critical parts of the application [26].
- A high-level annotation language for coordinating PEPPHER components and expression of parallel pipeline patterns [28].
- A runtime system able to use performance information and efficiently schedule components across available hardware resources of a heterogeneous many-core system.

An important characteristic of the PEPPHER component model is the separation of specification and implementation of the components [30, 28, 25]. PEPPHER components may have multiple implementation variants providing the same functionality in a different way. Component implementation variants might be taken from an external library, or written and supplied directly by an "expert" (as opposed to mainstream) programmer and tailored for different parts of a heterogeneous many-core system. In this sense, PEPPHER provides a flexible and extensible model that enables a description of new or existing applications as a set of interacting, multi-architectural components.

Although the approach itself is essentially language independent, the current version of the PEPPHER framework supports C/C++ as the base language. The PEPPHER coordination language provides a set of non-intrusive language constructs in the form of C/C++ `#pragma` directives. It provides the means to identify the parts of the software that should make use of PEPPHER components, as well as to coordinate their execution. The PEPPHER approach utilizes task-based programming model, in which tasks correspond to PEPPHER components. The PEPPHER heterogeneous runtime system (StarPU [23, 30]) takes care of variant selection and efficient execution of the component tasks on the underlying hardware, while taking into account task data dependencies, data transfer costs between distinct memories of heterogeneous architectures, as well as performance models generated automatically from historic performance data. Figure 2.1 shows the PEPPHER software stack. This high-level overview of the framework, illustrates how different fit together.

2.2 Component Model

The PEPPHER component model builds on well-established component based system engineering (CBSE) concepts [31][25], which are further extended by the association of meta-data

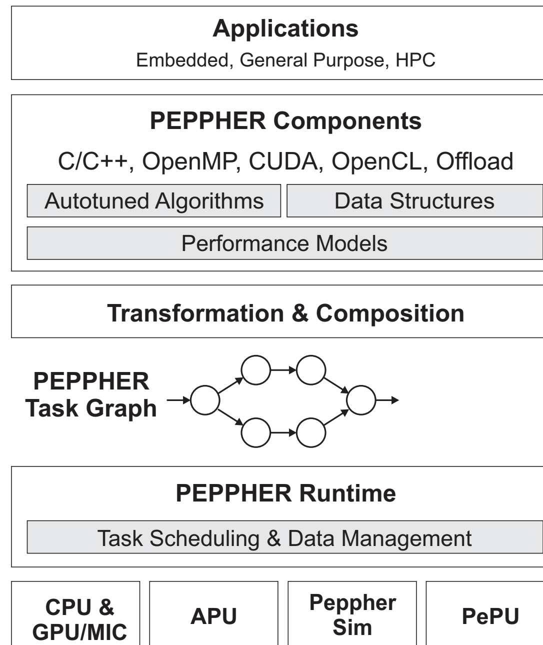


Figure 2.1: PEPPHER software stack [27]. PEPPHER applications are currently written in C/C++ and made *performance-aware* by turning performance-critical parts into performance aware components with multiple implementation variants. The components may be parallelized internally, using conventional parallel programming languages. Transformation and composition techniques aid the generation of component implementation variants which are scheduled and dynamically selected for execution on available resources by the runtime system.

with the software components. A PEPPHER *component* is defined as an annotated software module that implements specific functionality [25]. Essentially, a PEPPHER component consists of an abstract component interface and of one or more implementation variants (see Figure 2.2).

In the forthcoming sections we will describe the interface and the implementation variant meta-data more closely. Additionally, we demonstrate relevant parts of the framework using *DetectFaces*, a multi-architectural component from the image processing example (Face Detection), which is evaluated in Chapter 6. The component uses CPU and GPU implementations re-engineered from the OpenCV library [32] both providing the same functionality of detecting human faces on an image and drawing rectangles around them. Note that this documents only describes relevant parts of the full specification and XML schemas that can be found in the PEPPHER Deliverables [25, 28].

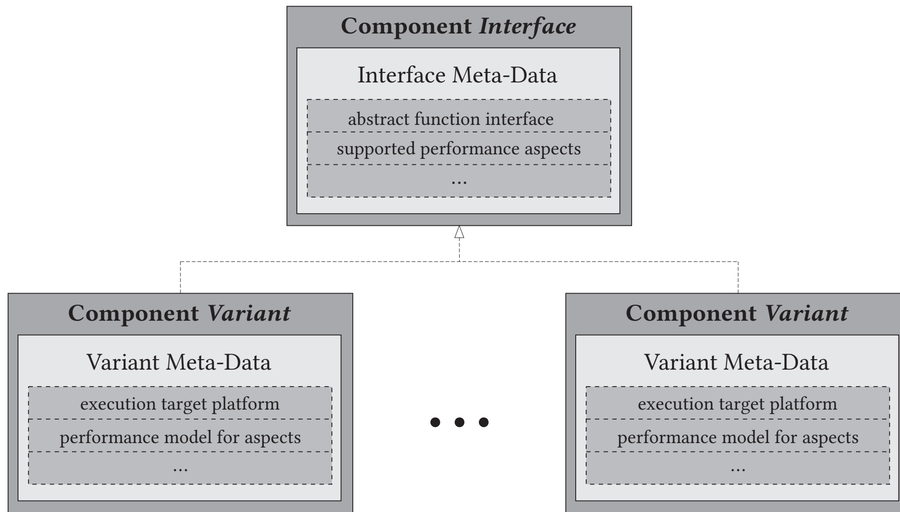


Figure 2.2: PEPPHER components consist of an abstract interface and one or more implementation variants [28].

As mentioned above, the PEPPHER component model allows storage of component meta-data which is later used by other tools for making optimization and execution decisions. Each implementation variant, as well as the component interface have its own meta-data in order to capture relevant non-functional component properties and enable selection of the best appropriate implementation for underlying hardware or performance criteria. Typically, meta-data includes information on data accessed by a component, performance prediction information and platform specific resources for which a specific component implementation has been written. The meta-data is either provided by an expert programmer or generated automatically (by static analysis, profiling, etc.). In the current prototype, PEPPHER supports applications written in C/C++ and the identification of the components is accomplished through C/C++ pragma directives and it is further described by the external XML-based descriptors [27].

2.2.1 Component Interface

The PEPPHER component interface shares some similarities with public function prototypes or method interfaces in C/C++. In fact, a PEPPHER *component interface* specifies only the function declaration [25], but not its implementation. Listing 2.1 shows a component interface example.

```
// face detection interface
void DetectFaces(uchar *img_data, unsigned int img_size);
```

Listing 2.1: *DetectFaces(...)* - non-generic component interface.

The PEPPHER interface is fully defined by the interface descriptor, which is an XML document storing the function name, parameter type, access types (read, write, read/write) and information on performance capabilities, for which performance aspect a component is to be optimized [25, 28]. The XML schema comprises the following elements:

- The Component Interface Identifier - a unique name for the interface, stored in the *name* attribute of the *Interface* element.
- The Interface Parameter specification - stored in *parameters* element, it allows a specification of the name and type of the parameters, as well as the return type and the parameter access modes. For each parameter, in the *parameter* element, the access mode (read, write, read/write) can be specified explicitly with the *accessMode* attribute. The access mode attributes are crucial for the runtime system to be able to describe data dependencies between the tasks and achieve efficient runtime scheduling and execution decisions [25].
- Other/Optional parameters - parameters such as Interface *description*, *genericParameters*, *role* and *reference* are optional and used only in certain scenarios. For instance, reference of the parameter can be used to set a relation between two parameters.

A corresponding interface descriptor for the *DetectFaces* component is shown in the Listing 2.2.

```
<peppher:component>
  <peppher:interface name="DetectFaces" kind="non-generic"
                     id="2a5c14c0b7cfce31299026ff24baf6f3d0c9431a">
    <peppher:description>
      Detect faces in the image and draw the rectangles around detected faces.
    </peppher:description>
    <peppher:parameters>
      <peppher:parameter name="img_data" type="uchar*" accessMode="readwrite"/>
      <peppher:parameter name="img_size" type="int" accessMode="read"
                         role="Vec_nx" reference="img_data"/>
    </peppher:parameters>
```

```
    ...
  </peppher:interface>
</peppher:component>
```

Listing 2.2: The PEPPHER component interface descriptor for an image processing (face detection) component. The example shows a component with two related parameters. Parameter *img_size* describes the size of the 1D array *img_data*.

2.2.2 Component Implementation

The functionality declared by the component interface, may be implemented by multiple implementation variants, which opens up various possibilities. On the one hand, implementations can be used in combinations or interchangeably, selecting the best match for the particular target architecture or for the desired optimization goal. For instance, implementation variants may be tailored for different architectures, different algorithms or different input data distributions. Every component implementation has meta-data associated with it. The parent element for the component implementation, as described in XML schema, is the *implementation* element. It may contain the list of source files and instructions on how to compile them, including desired compiler version and command line switches, provided functionality, information on target platform which links it to an abstract platform model written in PEPPHER Platform Description Language (PDL) [25].

Listings 2.3 and 2.4 demonstrate component implementation variant descriptors. Both listings include the *implementation* element which stores the *name* and the *id* of the implementation.

```
<peppher:component>
  ...
  <peppher:implementation name="DetectFaces_cpu"
                          id="7ef1c109b6b7ef1835c841ef9d54f982c8a267c1">
    <peppher:description>
      Detect faces in image and draw rectangle. Implemented for CPU.
    </peppher:description>
    <peppher:providedInterfaces>
      <peppher:providedInterface id="2a5c14c0b7cfce31299026ff24baf6f3d0c9431a"/>
    </peppher:providedInterfaces>
    <peppher:sourceFiles>
```

```
        <peppher:sourceFile name="detectfaces_cpu.cc" version="0.1" language="c++">
          <peppher:compilation>
            <peppher:compiler>
              <peppher:name>gcc</peppher:name>
              <peppher:version>4.4</peppher:version>
              <peppher:flags>-O3</peppher:flags>
            </peppher:compiler>
            <peppher:language>c++</peppher:language>
          </peppher:compilation>
        </peppher:sourceFile>
      </peppher:sourceFiles>
      <peppher:targetPlatform name="cpu" progModel="serialc">
        <peppher:pdlRef>serialc_cpu.pdl</peppher:pdlRef>
      </peppher:targetPlatform>
    </peppher:implementation>
    ...
  </peppher:component>
```

Listing 2.3: The PEPPHER component implementation descriptor - the CPU variant of the *DetectFaces* component.

The optional *description* element is used to give useful information on what a component functionality is. The subelement *providedInterface* of the *providedInterfaces* refer to the *id* of the original interface descriptor shown in Listing 2.2, hence relating the implementation descriptors to the component interface descriptor. The *sourceFiles* element contains the list of the source files of the component implementation variants, as well as the rules how to compile them.

The CPU implementation variant shown in Listing 2.3 specifies gcc compiler version 4.4 with "-O3" optimization switch to be used for the compilation of "detectfaces_cpu.cc", which is written in C++. Since GPU variant for this particular example utilize the OpenCV [32] library, the CUDA compiler is not required for the compilation of the GPU variant. Instead OpenCV provides interfaces for invoking the CUDA calls from regular C/C++ code.

```
  <peppher:component>
    ...
    <peppher:implementation name="DetectFaces_gpu"
                            id="3b9301143d6799c9d30fdaea3cdbf6d0ac5a73df">
      <peppher:description>
        Detect faces in image and draw rectangle. Implemented for Nvidia CUDA.
      </peppher:description>
      <peppher:providedInterfaces>
        <peppher:providedInterface id="2a5c14c0b7cfce31299026ff24baf6f3d0c9431a"/>
```

```
      </peppher:providedInterfaces>
      <peppher:sourceFiles>
        <peppher:sourceFile name="detectfaces_gpu.cc" version="0.1" language="c++">
          <peppher:compilation>
            <peppher:compiler>
              <peppher:name>gcc</peppher:name>
              <peppher:version>4.4</peppher:version>
              <peppher:flags>-O3</peppher:flags>
            </peppher:compiler>
            <peppher:language>c++</peppher:language>
          </peppher:compilation>
        </peppher:sourceFile>
      </peppher:sourceFiles>
      <peppher:targetPlatform name="cuda_gpu" progModel="cuda">
        <peppher:pdlRef>cuda_gpu.pdl</peppher:pdlRef>
      </peppher:targetPlatform>
    </peppher:implementation>
    ...
  </peppher:component>
```

Listing 2.4: The PEPPHER component implementation descriptor - the GPU variant the *DetectFaces* component.

In contrast to *serialc* programming model, Listing 2.4 shows *cuda* as the required model for GPU variants. In this example, such set-up will instruct the runtime system to handle the data transfers between host and GPUs , overriding the default OpenCV mechanism.

2.3 Execution Model

PEPPHER aims at exploiting multiple levels of parallelism. On the one hand, there is intra-component parallelism found in the component variants, parallelized using different programming models and bound to a particular model given by specified platform description (e.g. data parallel CUDA or OpenCL component implementations). On the other hand, the PEPPHER execution model is based on a task-parallel model, where components are invoked asynchronously and executed in parallel by corresponding tasks [28]. Usually referred to as inter-component parallelism, this mechanism relies on the runtime system to select an appropriate component variant and orchestrate parallel execution of components. In fact, at runtime applications are represented with a DAG. The vertices (nodes) of the graph represent the tasks executing component implementations, while the directed edges represent the data

dependencies between the tasks. Figure 2.3 outlines the PEPPHER execution model.

There is a distinction between the master process and the worker processes within the model. The master process is concerned with the initiation of the PEPPHER framework, it manages and maps components function calls to workers and terminates the execution [28]. On the other hand, the worker processes are characterized by the use of the different execution units or access different memory spaces. The PEPPHER runtime system takes care of selecting the component implementation variants and its assignment to a suitable worker.

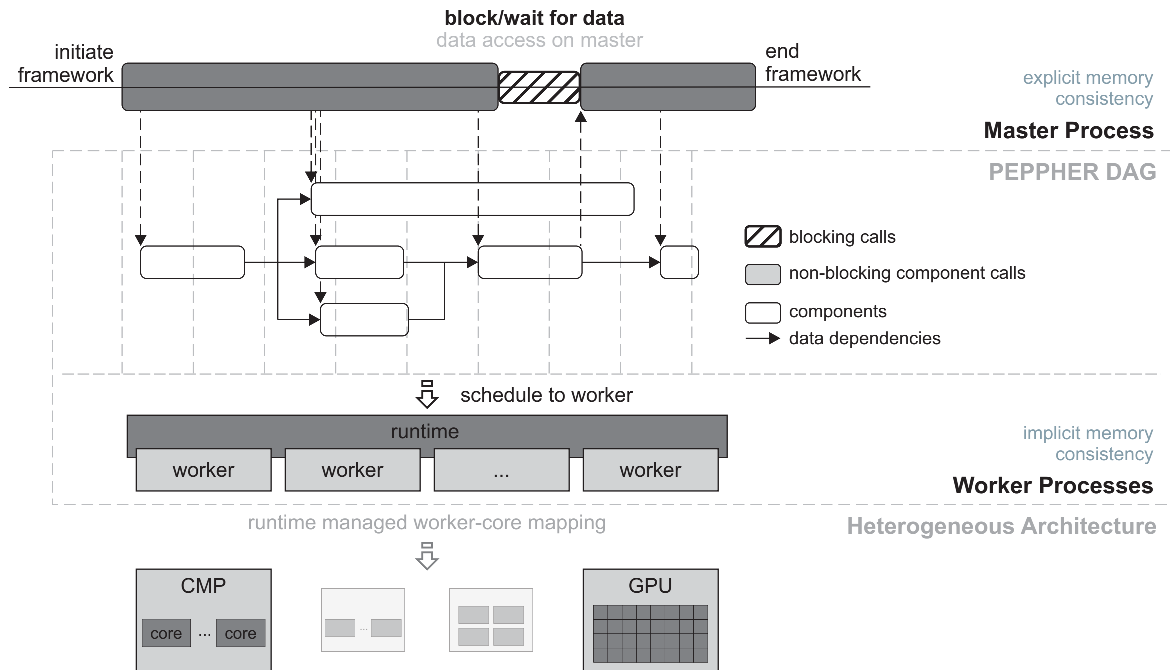


Figure 2.3: PEPPHER execution model [28].

Such configuration allows the runtime system to make better decisions. Selection of component implementation variants and their assignment to a suitable worker considers system occupancy, estimated performance, execution history, input data availability, data transfer costs and performance optimization criteria. Since the application registers its input and output data with the runtime, the system is able to transparently handle the transfer of the data to workers and keep the data consistent on the worker process level. Additionally, data awareness enables the runtime to make better scheduling decisions including the information on the cost of the data transfer. To ensure data consistency on the application level, master process must issue *flush* command to ensure the data is up to date. Note that possible intra-component data dependencies may delay execution until the input data is ready. Further on, certain computations may benefit from splitting the work among more workers.

Therefore, sometimes it proves beneficial to map single component invocation to multiple runtime subtasks, which can be mapped to different execution units and process different part of data in parallel [33].

2.4 Coordination Primitives

The PEPPHER framework provides a coordination language in the form of annotations, which is used in the PEPPHER application on the inter-component level [28]. It aims to support incremental transformation of existing applications for execution on hybrid architectures. Basic coordination primitives support synchronous and asynchronous calls to components. Asynchronous calls perform non-blocking invocation of the components, thus potentially enabling parallel execution. An actual call to a component will instruct the framework to schedule a suitable implementation variant. However the time of the execution will depend on the actual scheduling policy, dependencies and resource availability. In contract, synchronous calls will block until the execution of the component is over. The *call* annotation represents a basic construct for invoking components. The default behaviour is to call component-provided functionality asynchronously. However, such behaviour might require explicit synchronization which is addressed with the *flush* construct. We summarize clauses that can be used with the *call* construct in the following list:

- *sync* clause instructs the framework to execute subsequent component function call synchronously
- *parameter* clause enables parameter assertions
- *optimize* clause allows user to specify the optimization goal
- *target* clause allows user to specify the preferred execution target
- *partition* clause allows user to provide data partitioning information, assuming input parameters in the component function calls are arrays
- *access* clause enables data access patterns assertions within the component function call

The *call* construct is used in the source code of the original application. Currently, C/C++ applications are supported and coordination constructs are provided in the form of *#pragma* directives. Figure 2.5 shows the syntax of the *call* construct.

Syntax

```
#pragma pph call [clause,[[,]clause]...]
{
   component function call
}
where clause is one of the following:

sync
parameter(scalar-logical-expression-for-interface-parameters)
optimize(performance-information)
target(target)
partition(array-partition-spec [[,] array-partition-spec] ...)
access(array-access-spec [[,] array-access-spec] ...)
```

Listing 2.5: Syntax of the PEPPHER *call* Construct [28].

Semantics

The default behavior of the *call* construct will mark the corresponding component to be called asynchronously. Listing 2.6 demonstrates the use of the call construct.

```
#pragma pph call
DetectFaces(img_data, img_size);
```

Listing 2.6: *call* construct: asynchronous component call.

If followed by a sync clause, the component call will be synchronous (see Listing 2.7)

```
#pragma pph call sync
DetectFaces(img_data, img_size);
```

Listing 2.7: *call* construct: synchronous component call.

In case where data dependencies between two consequent component calls exist, the execution of the dependent call will be postponed by the runtime system until the data is ready.

Listing 2.8 depicts such scenario. Both components will be scheduled asynchronously for execution, however the execution of the *DetectFaces* component will start once the task running the *ConvertAndResize* function has been terminated.

```
#pragma pph call
ConvertAndResize(in_img_data, in_img_size, img_data, img_size);
#pragma pph call
DetectFaces(img_data, img_size);
```

Listing 2.8: *call* construct: asynchronous call to the *DetectFaces* will be executed once *ConvertAndResize* finishes.

In general, memory consistency in the application is not guaranteed. If necessary, it must be handled explicitly with the *flush* construct (see 2.9).

```
#pragma pph call
ConvertAndResize(in_img_data, in_img_size, img_data, img_size);

// flush is not needed here because of the
// implicit coherency across the workers
#pragma pph call
DetectFaces(img_data, img_size);

 // block until img_data has become available
#pragma pph flush(img_data)
... // do something with img_data
```

Listing 2.9: *flush* construct: block until requested data has become available.

The *parameter* clause enables specification of assertions for component calls. The current prototype supports only scalar parameters. The Listing 2.10 shows component function call with restricted size of the input image data array.

```
// make a call to component,
// but restrict the grayscale image size to specific resolution
#pragma pph call parameter (img_size == (1024*768) )
DetectFaces(img_data, img_size);
```

Listing 2.10: *call* construct: *parameter* clause enables assertion for component parameters.

Optimization of a PEPPHER application can be explicitly influenced via the *optimize* clause. This way, users can give priority to a certain aspect of performance. Listing 2.11 gives an example of the prioritization of the execution time. A a consequence, only the component implementation variants that support specified performance aspects will be considered during component selection [28].

```
#pragma pph call optimize (EXECUTION_TIME)
DetectFaces(img_data, img_size);
```

Listing 2.11: *call* construct: *optimize* clause allows prioritization of desired aspects of performance .

Finally, the *target* construct enables setting up the preferred execution target for a component call. The target can be either pre-defined library (such as CUDA or OpenCL), or processing unit group as defined in platform descriptor [25]. Listing 2.12 shows an example for the *target* clause.

```
#pragma pph call target (CUDA)
DetectFaces(img_data, img_size);
```

Listing 2.12: *call* construct: *target* clause allows user to influence variant selection by setting up the desired target platform or library.

The PEPPHER coordination language also provides coordination primitives for higher-level parallel design patterns (such as pipelining and task farming), which aim to improve programmability of heterogeneous multi-cores. The coordination primitives for pipelining are highly relevant for this work, therefore they are explained Section 3.3.

3 Pipeline Pattern

3.1 Introduction

Pipelining is often compared to the well known process of an assembly or a production line. The idea of an assembly line is that the process of assembly of a product can usually be broken down into a series of steps (stages) that perform different operations, for instance installing different parts of a vehicle at each step as it moves down the assembly line. Each worker (or a group of workers) is responsible for a single step of the process, which is applied to every item that comes on the assembly line. This allows overlapping of the work on individual stages of the assembly process. If the number of vehicles to be produced is sufficiently high, utilization of resources and workforce should be more efficient.

As considered in this thesis, a pipeline consists of several interconnected pipe stages, which processes a stream of data flowing though it [34, 35]. In essence, data enters the pipeline, goes through all stages and exists at the other end. Such approach potentially brings cheap parallelism opportunities for certain codes, especially considering the work on large amounts of data that needs to be processed in several stages, making it particularly useful in the cases where parallel loops can not be used due to the data dependencies and the data needs to be processed in sequential order. In fact, the ultimate goal of pipelining is not to speed up the execution of a single task, but to increase overall throughput. Consider the following while loop:

```
while ( !end ) {
  S1(data, input);
  S2(data, processed_data);
  S3(processed_data, output);
}
```

Listing 3.1: While loop performing 3 different operations on the same data.

Let *S1*, *S2* and *S3* be the steps in the pipeline, called *pipeline stages*. Let each stage operate as follows:

- consume input data
- processes the data
- emit the data

Data flowing through this *while loop* must go through all three stages in order enforced by the data dependencies (note that the same data is used by more then one stage). Sequential, non-pipelined execution of this loop is illustrated in Figure 3.1.

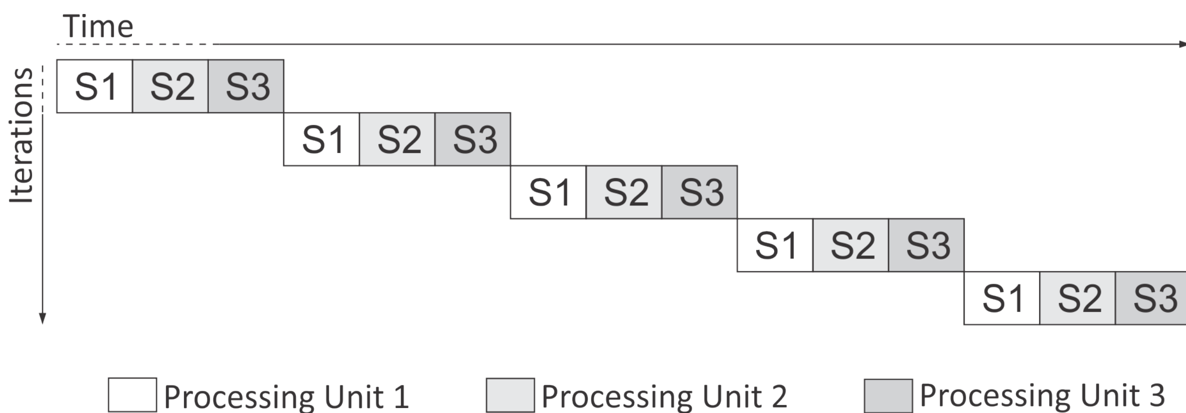


Figure 3.1: Non-pipelined execution of five iterations. Since the while loop executes sequentially, processing units may be in idle state while the iteration is over.

We distinguish between intra- and inter-stage parallelism. *Inter-stage parallelism* refers to the parallelism between the pipeline stages - if the stages are mapped to different processing units, CPU cores for example, different stages can process different data in parallel. *Intra-stage* parallelism outlines the possibilities to use the parallelism inside the stage and eventually execute it on multiple execution units for example on an accelerator (e.g. GPU, Intel Xeon Phi). Further on, if we assume that we have enough input data and that the while loop has enough iterations, overlapping iterations in such pipeline-like manner could bring further benefits. For example, consider the same while loop executed in a pipelined fashion, where each processing unit processes one stage, assuming there are no loop-carried dependencies from *S2->S1* or *S3->S1* can lead to execution described in Figure 3.2.

Finding the optimal configuration and the perfect overlapping defines the characteristics of the pipeline such as *throughput*, *latency*, *depth* and *stage length*. *Throughput* of a pipeline

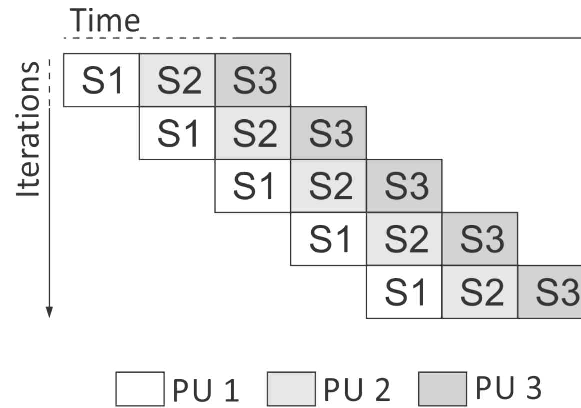


Figure 3.2: Pipelined execution. This figure depicts ideal pipeline execution where there are no loop-carried dependencies.

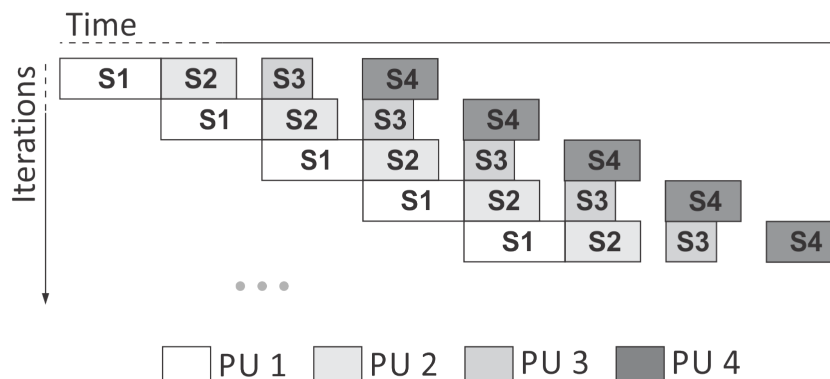


Figure 3.3: Pipelined execution. This figure illustrates a situation where stages have uneven lengths. Such a scenario may have negative influence on the potential speed-up.

is a measure of the number of data items it can process per time unit. *Latency* is defined as an amount of time between input generation and actual completion [2]. Depending on the problem at hand, latency or throughput may be prioritized. On the one hand, very deep pipelines can introduce high latency. On the other hand, shallow pipelines may lack available parallelism. A pipeline is fully utilized when all stage are active. Full utilization depends on its depth, which relates to the number of the stages. An ideal pipeline has all stages of equal length, as shown in Figure 3.2. However, this is rarely the case and uneven stage length often leads to a scenario where a pipeline is imbalanced as illustrated in Figure 3.3.

Figures 3.2 and 3.3 immediately reveal several aspects of pipelining. We summarize them in the following list:

- Overlapping the execution of stages potentially enables parallelism as multiple stages

may operate simultaneously.

- Some overhead is introduced until a pipeline is filled so that all computational units are busy. Similarly, there is an overhead when a pipeline is drained and all computational elements have completed executing. This may influence the potential speed-up in case of deep and short pipelines.

- Increase in number of stages increases the potential speed-up. However, imbalanced stage lengths may have an impact on the performance.

- Depth of a pipeline and balance of stages determine *throughput* and *latency* of the pipeline.

Often, balancing of stages is not an easy task. However, exploiting such possibilities on new heterogeneous hardware could bring further benefits. For instance, running compute-intensive, long running stages could be passed to an accelerator for faster execution, therefore improving the balance of a pipeline.

Besides depth and length, pipelines have other important structural properties. We differentiate between linear and non-linear pipelines and between buffered and non-buffered pipelines. In linear pipelines, every stage, except source and final stage, have their successor(s) and predecessor(s), whereas non-linear pipelines may have an arbitrary configuration.

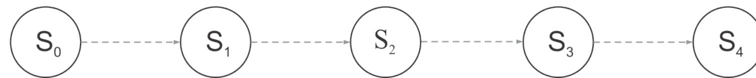


Figure 3.4: Five stage linear pipeline.

Figures 3.4 and 3.5 demonstrate the difference between linear and non-linear pipelines. Although more complex, non-linear pipelines may open up additional parallelism possibilities, where certain stages may contain operations that can execute in parallel. Pipeline stages can make use of buffer structures that are used to feed data into stages. Such buffered pipelines allow decoupling of stages and mitigation of performance differences [34].

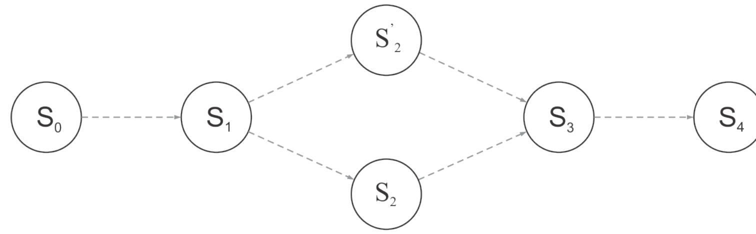


Figure 3.5: Five stage non-linear pipeline.

3.2 Performance Considerations

In the literature, it is often considered that on a homogeneous system, a pipeline pattern works generally better if the operations performed by various stages of a pipeline are equally compute-intensive [2] - that is if a pipeline is well-balanced. In contrast, in imbalanced pipelines, the longest running stage determines and limits the potential speedup. However, if the execution units differ, as is the case with heterogeneous systems, mapping compute intensive stages to faster execution units, such as accelerators, can potentially improve the pipeline balance and speed up the whole pipeline.

Other source of performance degradation comes from the fact that the pipeline needs to be filled and drained. In fact, the full benefit of pipelining is not reached until the pipeline is filled and fully utilized. Therefore, pipelines usually require significant workload to amortize the fill and drain overhead.

Implementing pipelines in terms of tasks executing on different execution units of a heterogeneous many-core requires careful management of the data flow. Due to the different memory access latencies and transfer rates, the data flow is usually required to be buffered [2]. Although buffered pipelines are often more memory demanding, buffers provide more flexibility in terms of execution control and handling of compute-intensive stages.

The data flowing through the stage is consumed via stage input ports and emitted via stage output ports. Multiple input/output stage ports enable finer control of data passed between pipelines stages, in contrast to aggregation and disaggregation in-between the stages. Furthermore, the usage of buffers allows additional level of execution control. For instance, capacity limited buffers may be used to control the memory footprint, limit the degree of parallelism in order to improve the balance of the pipeline, or to provide performance tips for an intelligent runtime system.

The assignment of stages to processing units of hybrid hardware may not be an easy task. Load balancing and data management play a major role. On the one hand, pipeline stages need to be mapped to available execution units. However, in order to improve load balance, a runtime system must take into a consideration the current system load, available tasks and their respective compute intensity, as well as heterogeneity of the underlying hardware. In this context, mapping of compute intensive stages to faster processing units may be beneficial, as it may improve overall pipeline balance. On the other hand, minimizing the movement of data between different memories of heterogeneous system and improving locality in complex pipelines may bring significant performance benefits.

3.3 Pipeline Directives

This section addresses high-level language support for pipeline patterns in the form of C/C++ *#pragma* preprocessor directives. The description of the pipeline patterns via code annotations is addressed at two levels:

- at pipeline level, with the *pph pipeline* construct - allowing global pipeline configuration, such as default size of buffers between stages, replication factors, etc.
- at stage level, with *pph stage* construct - overriding global pipeline configuration and allowing adjustment of individual stage parameters

3.3.1 *pipeline* Construct

Annotating a while loop with the PEPPHER *pipeline* construct indicates that it should be transformed into a parallel pipeline. The syntax for such annotations is given in Listing 3.2:

Syntax

```
#pragma pph pipeline [clause [[,] clause]]
while(while-exit-condition) {
    component function call
    ...
    component function call
}
```

```
where clause is one of the following:

buffer(buffer-type, buffer-size)

and where buffer-type is one of the following:

UNORDERED: no ordering
FIFO: first-in-first-out buffer
PRIORITY: priority buffer
```

Listing 3.2: Pipeline coordination construct.

Semantics

Typically, in the context of PEPHER, the while loop contains functions calls - interfaces to the multi-architectural PEPPHER components. Unless overridden by the *stage* construct (see Section 3.3.2), the default behaviour is considering the components to be pipeline stages, whose input and output parameters (stage ports) are further analyzed in order to determine the structure of the pipeline. Finally, the while loop exit condition needs to be evaluated to a valid boolean value, since it translates to the termination criteria of the pipeline. Enabling a transformation to a parallel pipeline for an arbitrary while loop comprised of PEPPHER components, with all the default settings is shown in Listing 3.3.

```
#pragma pipeline
while ( ... ) {
  // component interface calls ...
}
```

Listing 3.3: Simple example of the *pipeline* construct. Annotating a while loop in this way enables transformation to a parallel pipepline with the default values.

Currently, we support buffered pipelines. This implies that the default mechanism will automatically insert buffers between all inter-connected pipeline stages. Buffer settings can be fine-tuned by including the *buffer* clause. This clause allows selection of different buffer ordering policies.

```
#pragma pipeline buffer(UNORDERED, 16)
while ( ... ) {
  // component interface calls ...
}
```

Listing 3.4: Pipeline construct. Stages will be connected using *unordered* buffers of size sixteen.

We support *unordered*, *fifo* and *priority* buffers. Listing 3.4 depicts a scenario where *fifo* buffer, with capacity of sixteen elements is used for all stages in the pipeline.

3.3.2 *stage* Construct

Each stage in the pipeline may have accompanying *#pragma* that serves to override the default settings. The syntax of the *stage* construct in the current prototype is shown in Listing 3.5.

Syntax

```
#pragma pph stage [clause [[,] clause]...]
{
    component function call
    ...
}
where clause is one of the following:

buffer(buffer-type[[,](buffer-size)])
replication(replication-count)
```

Listing 3.5: Stage coordination construct.

Semantics

The *stage* construct allows fine tuning of pipeline stages. We support adjustment of stage replication factors, stage merging and specification of buffer types and sizes. The merging of stages allows calls to more than one component within a stage. This is accomplished by means of curly braces (“{” and “}”) as shown in Listing 3.6.

```
#pragma pipeline
while ( ... ) {
  stage_1( ... );
  #pragma stage {
    stage_2( ... );
    stage_3( ... );
  }
  stage_4( ... );
  ...
}
```

Listing 3.6: Stage construct: Stage merging - curly braces signify that stage_2 and stage_3 will be merged.

Buffer types and sizes can be modified for each stage. Note that explicitly changing buffer types overrides the default or global pipeline settings, which may influence the behavior and the correctness of an application in certain scenarios. In fact, due to the parallel execution of the replicated pipeline stages, *unordered* buffers may influence the order of data packets arriving to succeeding stages. While this may bring some performance benefits, it should be used with caution. If strict ordering of data packets is required, than different buffer types should be used later in the pipeline in order to preserve correctness. Usage of the *buffer* clause is shown in Listing 3.7.

```
#pragma pipeline
while ( ... ) {
  ...
  #pragma stage buffer (UNORDERED, N*2)
  stage_n( ... );
  ...
}
```

Listing 3.7: Stage construct: Adjusting the queuing mechanism for stage_n buffer.

The *replication* clause may be used to adjust the replication factor of a stage. This directs the framework to replicate the annotated stage object. Since each stage runs in a separate thread, replication enables multiple independent data packets to be submitted to runtime system and eventually processed in parallel. Unless this parameter is to be tuned automatically, a programmer should be aware of all performance considerations and possible bottlenecks that might arise (e.g., replicating stage excessively could lead to thread over-subscription and possible performance degradation). Listing 3.8 demonstrates how to set stage a replication factor.

```
int rf = 4;
#pragma pipeline
while ( ... ) {
  ...
  #pragma stage replication (rf)
  stage_n( ... );
  ...
}
```

Listing 3.8: Stage construct: Replication factors.

Finally, combined usage of constructs is demonstrated in Listing 3.9.

```
...
int rf = 4; // replication factor
#pragma pipeline
while ( ... ) {
  stage_1( ... );
  // merge stage_2 and stage_3, use unordered buffer
  // and set replication factor
  #pragma stage buffer (UNORDERED, N*2) replication (rf) {
    stage_2( ... );
    stage_3( ... );
  }
  // connect to stage4 stage with PRIORITY
  #pragma stage buffer (PRIORITY, N*2)
  stage_4( ... );
}
...
```

Listing 3.9: Stage construct: Combined usage of pipeline directives.

4 Coordination Layer

The coordination layer for pipelining supports multithreaded parallel execution of the pipelines on heterogeneous many-core architectures with the goal of utilizing all existing execution units simultaneously. Internally, the layer utilizes StarPU [23] heterogeneous runtime system that is responsible for dynamically selecting suitable component implementation variants for pipeline stages and for scheduling their execution to different execution units of a heterogeneous many-core system in a performance and resource efficient way. This interplay between the pipeline coordination layer and the underlying runtime system is a very important aspect of this work, thus it will be elaborated here in more details.

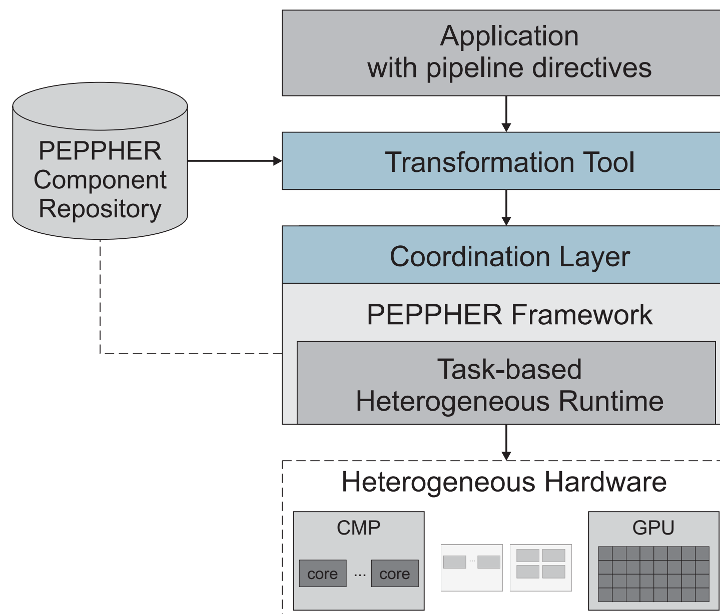


Figure 4.1: Overview diagram of the PEPPHER framework [29]. On the top of the stack are applications that make use of the PEPPHER multi-architectural components and feature pipeline-annotated while loops. These applications are processed and transformed by the transformation tool into a C++ code that utilizes the pipeline coordination layer for managing parallel execution on heterogeneous many-core systems (also known as hybrid systems).

4.1 Overview

The coordination layer for pipelining has been developed as a part of PEPPHER programming framework. It interacts with the component framework, the transformation tool, and the task-based heterogeneous runtime system (StarPU [23]). Figure 4.1 shows a high-level overview of the framework.

In PEPPER a pipeline realization is based on corresponding annotated performance critical parts (while loops) in the source code of applications from various domains. A possible scenario is depicted in Listing 4.1.

```
...
int rf = num_available_exeution_units(); // replication factor
#pragma pph pipeline buffer ( PRIORITY, N*2 )
while ( inputstream >> file ) {
  func1 ( file, data ); // connect func1 to func2 via data
  #pragma pph stage replicate ( rf ) // adjust replication factors
  func2 ( data, cdata ); // connect func2 to func3 via cdata
  func3 ( file, cdata );
}
...
```

Listing 4.1: High-level code annotated with PEPPHER pipeline directives.

Stages of such pipeline relate to function calls inside the while loop. The function calls correspond to the PEPPHER multi-architectural components and are in fact the component interfaces. Each component may have multiple implementation variants targeting different processing units of heterogeneous system (see Chapter 2). During the transformation process, the framework interacts with the PEPPHER component repository by selecting available component implementation variants and integrating them into the transformed code (see Chapter 5). During the execution, the component implementation variants are then delegated to the coordination layer and ultimately to the runtime system. Based the component information, user-provided information and code analysis, the transformation system produces a C++ code with calls to the coordination layer for pipelining. The layer utilizes a heterogeneous runtime system to schedule pipeline stages for parallel execution on the appropriate execution units. The current prototype supports the StarPU heterogeneous runtime system [23] that is able to transparently handle data management and perform intelligent task scheduling.

The coordination layer for pipelining is the main contribution of this work. It is an object-oriented C++ library comprising a number of classes that address structural and functional properties of pipelines, stages and buffers, as well as the management and runtime classes, which in turn address the coordination aspects of the layer. Even though the coordination layer can be used directly by a programmer, we have developed a source-to-source compiler implemented with the ROSE compiler framework that takes care of automatic transformation of the annotated high-level code to the code utilizing the layer (see Chapter 5). In fact, the coordination layer is an intermediate layer between programmers (which may construct an application using the high-level coordination language provided by PEPPHER) and the runtime system. Therefore, all methods described in this chapter are automatically generated with the parameters based on the user-provided information in the source code and the information from PEPPHER component repository. Following list summarizes relevant classes and their purpose:

- Stage class - parameters, ports, functionalities and execution coordination of the pipeline stages
- Buffer class - an abstract class for buffer management, data packet ordering and synchronization policies
- Pipeline class - structural properties of the whole pipeline, global pipeline settings and (re-)configuration methods
- Runtime class - an abstract class for unified interface to the underlying runtime system
- PipelineManager class - execution management of one or more pipelines
- Other classes - various helper classes, such as *StagePort*, *StageParameter*, *StageFunctionality*, *BufferElement* and the concrete class implementations derived from the abstract classes. These are *FifoBuffer*, *PriorityBuffer*, *LockFreeBuffer* and *StarPURuntime*.

In conformance with the PEPPHER objectives, this work also focuses on single node heterogeneous many-core systems that often comprise accelerator units, such as CUDA/OpenCL enabled graphic cards and/or Xeon Phi's in addition to the conventional multi-core processors.

4.2 Structure

The coordination layer is structured and used as a C++ library. It consists of four major parts that reflect the modular implementation of pipelines, buffer structures, runtime and profiling facilities. This is shown in the directory structure as well (see Listing 4.2). While the current version of the coordination layer (or “VPattern” library) implements only pipeline patterns, we plan to extend it to support other design patterns as well (e.g: master-worker, map-reduce). Therefore, the layer is comprised of modular and reusable components separated in logical groups.

All facilities related to the buffer and queuing mechanisms are contained inside the “buffer” folder. Similarly, the pipeline and runtime facilities are contained within the “pipeline” and “runtime” folders. The coordination layer does not implement its own runtime system, but relies on an external one. Therefore, it provides a class that abstracts the required features that runtime needs to have, in order to be able to function together.

The library comes with two example applications featuring varying number of component implementation variants utilizing different programming models, such as OpenMP, OpenCL and CUDA that target efficient execution on different execution units. The first example is taken from the area of computer vision. In this example each image from a video sequence or a series of images is analyzed in order to detect human faces. The second example is from the area of bioinformatics and considers multiple pairwise alignment of many DNA sequence pairs using Needleman-Wunsch algorithm. Examples are elaborated in greater detail in Chapter 6 .

```
libvpattern
  - buffer
     - buffer.h
     - bufferelement.h
     - fifobuffer.h
     - lockfreebuffer.h
     - prioritybuffer.h
     - stdbuffer.h
  - common
     - vpatterncommon.h
     - vtttimer.h
  - examples
    + facedetect
    + nw
```

```
  - pipeline
    - pipeline.h
    - pipelinemanager.h
    - stage.h
    - stagefunctionality.h
    - stageparameter.h
    - stageport.h
  - profiling
    - datapacketmeasurement.h
    - stagemeasurement.h
    - stagestatistics.h
  - runtime
   - runtime.h
   - runtimeconfiguration.h
   - starpuruntime.h
  - vpattern.h
```

Listing 4.2: The structure of the coordination layer. Only the header files (*.h) are shown inside the directories.

4.3 Pipelines

In the coordination layer pipelines are represented by the *Pipeline* class that provides facilities for managing structural and functional aspects of pipeline patterns. The structural aspects address the general pipeline configuration, such as stage layout, configuration of the buffer structures that are inserted between the stages, default replication factors, pipeline-specific profiling info, as well as the methods for stage creation and linking (see figures 4.4, 4.8). The functional aspects address execution of the pipeline by providing *ExecuteAsync()* and *Wait()* methods that enable asynchronous pipeline execution, and *Pause()* and *Resume()* methods that allow the pipeline to be paused and resumed when necessary (e.g., when a certain pipeline or runtime properties need to be reconfigured). Important methods of the *Pipeline* class are summarized in Listing 4.3.

```
...
// consturctors
Pipeline();
Pipeline(BufferSyncPolicy);

// stage creation methods
Stage * CreateStage();
```

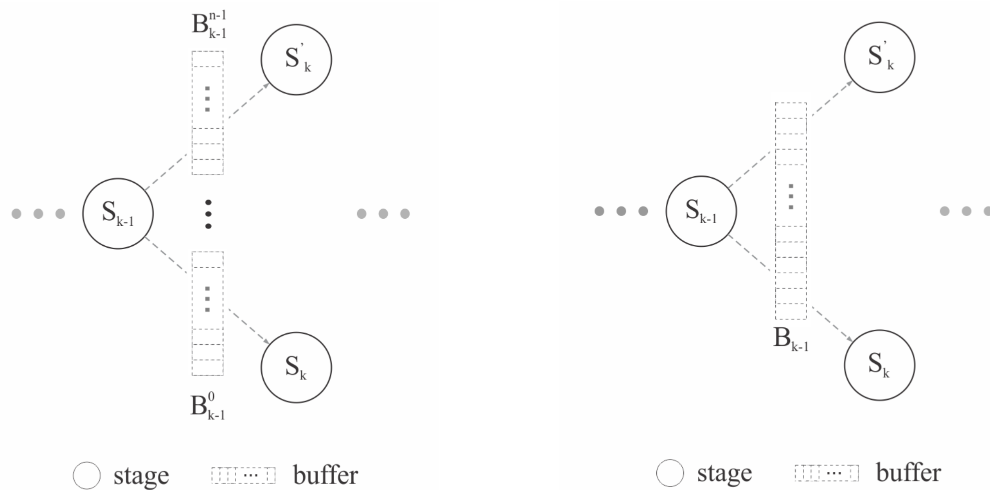
```
Stage * CreateStage(StageType);
Stage * CreateStage(std::string, StageType);

// connecting two stages
void ConnectStages(Stage *, Stage *);

// connecting stages overriding the default buffer sync policy
void ConnectStages(Stage *, Stage *, BufferSyncPolicy);

/* pipe control and synchronization */
void ExecuteAsync();
void Wait();

void Pause();
void Resume();
...
```

Listing 4.3: Important methods of the *Pipeline* class

The default settings for the global pipeline configuration assume a linear buffered pipeline with replication factor of one, the *priority* buffer ordering policy and lock-based queues (see section 4.5). The creation of stages is accomplished via the *CreateStage(...)* methods. Each method can be used to dynamically creates a *Stage* object, which lives in memory until the related *Pipeline* object is destroyed. The job of connecting stages is handled via the *ConnectStages(...)* methods that additionally instantiate the buffer objects based on the buffer creation policy in use. Buffer objects are created when stage connections are formed. Buffer creation policies determine the number and the position of buffers in the pipeline.

We support two such policies. The default is BUFFER_POLICY_AUTOMATIC_SINGLE, where the buffers are generated automatically in a such way that one single buffer is inserted between each two steps of the pipeline, regardless of the number of stage replicas or splits. This is shown in Figure 4.2. In contrast, BUFFER_POLICY_AUTOMATIC_MULTI inserts one buffer structure for each stage connection.

The *Pipeline::ExecuteAsync()* method will execute within a thread. This management thread will live on until the pipeline has finished executing. Essentially, this method iterates over all stage objects belonging to this pipeline and calls the *Stage::ExecuteAsync()* method that implements stage execution mechanism described in the next section.



(a) Multiple buffers between neighboring stages **(b)** A single buffer for all neighboring stages

Figure 4.2: Supported buffer creation policies. The left hand side describes the situation where a new buffer is insterted for each connected successor stage. The right hand side depicts the case where all connected stages share the same buffer.

4.4 Stages

A pipeline stage corresponds to a multi-architectural PEPPHER component. The stage, or the stage replica, is represented by the *Stage* class that includes its structural and functional information, such as stage type, parameters, ports, replication factors, information on the other stage replicas, functionality, directly connected stages and directly connected buffers for each stage port. As a stage object needs to belong to a pipeline object, it is usually created through the *CreateStage()* method provided by the *Pipeline* class (see Listing 4.4).

```
Stage *stage1 = pipeline.CreateStage();
Stage *stage2 = pipeline.CreateStage(STAGE_TYPE_SOURCE);
```

Listing 4.4: *Pipeline::CreateStage()* method will create a new stage object and automatically assign it to the pipeline

4.4.1 Stage Types

Every stage has a type, which relates to its relative position in the pipeline. The coordination layer distinguishes between three conceptual types of pipeline stages: *source*, *middle* or *sink*

stage. The stage type is implemented as a simple C++ enumerated type *StageType*. This is shown in Listing 4.5.

```
enum StageType {
  STAGE_TYPE_MIDDLE=0, // default
  STAGE_TYPE_SOURCE,
  STAGE_TYPE_SINK
}
```

Listing 4.5: *StageType* enum. Stage type relates to its relative position in the pipeline.

A *Source* stage is the first stage in the pipeline. As such it has no input buffers or predecessor stages. In addition to the execution of the associated functionality (e.g., component implementation variant), the source stage evaluates the condition of the *while loop* each time it is executed. Upon the transformation, the *while loop* condition is extracted and wrapped in the *WhileLoopWrapper()* function. The wrapper function is called and a boolean return value is evaluated in order to set the pipeline termination criterion, which is then propagated throughout the whole pipeline. Currently, the evaluation of the *while loop* exit condition is performed in a sequential manner, therefore enforcing the sequential queuing order of the output buffer packets from the *source* stage.

A *middle* stage, on the other hand, may have input and output buffers and queuing order based on the default or user settings.

Finally, a *sink* stage, has no output buffers or successor stages, since it is the last stage in the pipeline. *Sink* and *middle* stages do not require a particular buffer queuing order, unless specified by the user. By default, buffers use *priority* queues, which cover more general cases (see Section 4.5).

4.4.2 Stage Replicas

Stage replicas are clones of the original stage object. The only difference is the reference to the original object. In fact, every stage keeps references to all of its replicas, as well as to the original stage object. The default settings for a newly created stage object assume a stage with replication factor of one, with no connections to other stages. Listing 4.6 shows

relevant replication facilities provided by the *Stage* class.

```
...
// properties
  Stage * original_;
  std::vector<Stage *> replicas_;

// methods
...
  unsigned int replication_factor();
  void set_replication_factor(unsigned int replication_factor);
  unsigned int previous_replication_factor();
...
  void add_replica(Stage *stage);
  Stage * replica_at(int index);
  void remove_replica_at(int index);
...
  Stage * original();
  void set_original(Stage *stage);
...
  void CloneReplicas(Stage *source_stage);
...
```

Listing 4.6: *Stage* class replication facilities.

4.4.3 Stage Connections

The position of the stage in the pipeline defines the availability of its predecessor and successor stages (e.g., the *sink* stage will have no successors). The coordination layer for pipelining keeps track of all direct neighbors in two *std::vector* containers, as shown in Listing 4.7.

```
...
  std::vector<Stage *> predecessor_stages_;
  std::vector<Stage *> successor_stages_;
...
  std::vector<StagePort *> in_ports_;
  std::vector<StagePort *> out_ports_;
...
```

Listing 4.7: Stage object keeps track of stage connections and stage ports.

Essentially, stages are connected via ports. However, function calls (component interfaces) found in while loops may have many arguments that relate to the stage parameters. Stage ports are defined as collections of stage parameters. This particular feature is very useful when dealing with non-linear pipelines, where a stage may be connected to multiple stages via different combination of parameters - different ports. The current prototype provides the implementation of the stage *StagePort* class. The *StagePort* object is essentially a collection of *StageParameter* objects. Stage parameters are defined via the *StageParameter* helper class, which encapsulates the information on the parameter access modes, the element sizes and the number of elements in the data structure that should be handled.

```
pipeline.ConnectStages(stage1, stage2);
```

Listing 4.8: The simplest way of connecting stages via the *ConnectStages()* method of the *Pipeline* class.

4.4.4 Stage Functionality

The functionality of a stage is described by the *StageFunctionality* class. Stage functionalities relate to the component implementation variants. During the transformation process appropriate component implementations are queried from the PEPPHER component repository and included in the transformed code. Calls to component implementations are wrapped in stage wrapper functions. The *StageFunctionality* property of the *Stage* class contains a function pointer to the stage wrapper function and the information on the component implementation type, which refers to the type of the targeted processing unit that the particular component implementation is tailored for (e.g: CPU, CUDA, OpenCL).

4.4.5 Stage Execution

The stage execution is done in an asynchronous fashion via the *Stage::ExecuteAsync()* method - a wrapper function that spawns a new thread for each stage instance and invokes the *Stage::Execute()* method, which contains mechanisms for stage execution, coordination and synchronization. Essentially, each stage makes sure that the data handles are “popped” from the input buffers, posted for the execution to the runtime system and “pushed” to the output

buffers. This mechanism, shown in Figure 4.3, relies on the callback mechanism from the runtime system. Once the tasks have finished executing, the runtime system invokes the associated callback method that allows the stage instance to continue and eventually process the next data packet.

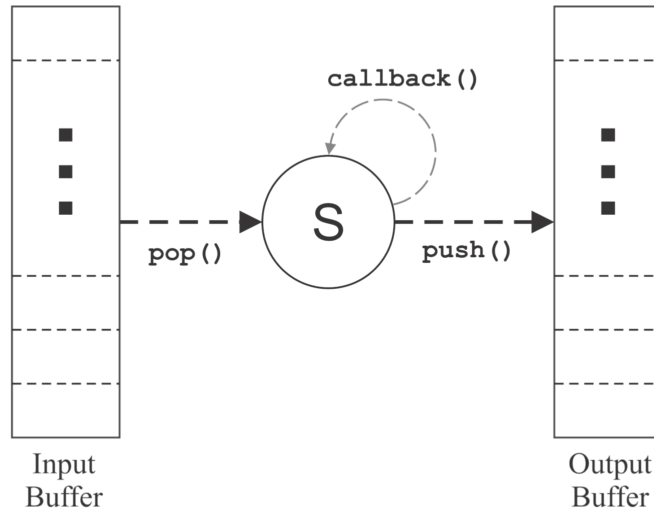


Figure 4.3: High-level description of the stage execution process. First, the buffer element is popped from the input buffer. When data is ready, the task is posted for execution and the data is pushed to the output buffer.

The stage replication mechanisms directly influence the degree of parallelism by providing a runtime system with enough potentially concurrent tasks to deal with. In addition, capacity bounded buffer structures serve as memory footprint regulators by limiting the maximum number of data packets that are to be held in memory at the time. Because of its relevance to this work, we dedicate a separate section (see Section 4.7) addressing this part of the work in greater details.

4.5 Buffer Mechanisms

The current prototype of the pipeline coordination layer utilizes buffered pipelines. This means that between every two stages a buffer is created automatically. The stages can insert (*push*) and remove (*pop*) data packets (or buffer elements) into/from the buffers. Due to the parallel execution of the pipelines, this approach allows different ordering of the data packets inside the buffers. The current prototype supports *fifo*, *priority* and *unordered* data packet ordering policies. Such mechanisms become useful when dealing with different kind

of algorithms. For instance, some applications may require strict ordering, such as video compression where sequential packet ordering must be imposed to preserve correct frame order [36], while others may benefit from non-strict ordering.

The coordination layer makes use of the data transfer mechanisms provided by the StarPU runtime systems. Hence, the layer does not perform the data transfers. Bounded (capacity-limited) buffer structures are used to setup and provide the runtime system with task handles used by the runtime to optimize data movement. As a consequence, the number of tasks that can be scheduled at a certain time is limited by the size of buffers.

Although this approach enables the runtime to optimize data transfers and execution of the tasks on heterogeneous systems, it also introduces some limitations when it comes to the control and profiling of the pipeline. More precisely, the tasks submitted to the runtime system are scheduled dynamically and the layer has no control over the execution order or the arrival order of the data packets produced by the stages. However, ordering of data packets can be forced by utilizing different queuing mechanisms for buffers (e.g. priority queues). In this case the layer makes sure that output buffers for replicated stages have minimum buffer sizes to allow such ordering.

The *Buffer* class is an abstract class generalizing the buffer access interface and is used to derive more specific buffer implementations. Currently, the coordination layer supports three derived buffer classes (*fifo*, *priority* and *unordered*) that utilize respective queuing mechanisms.

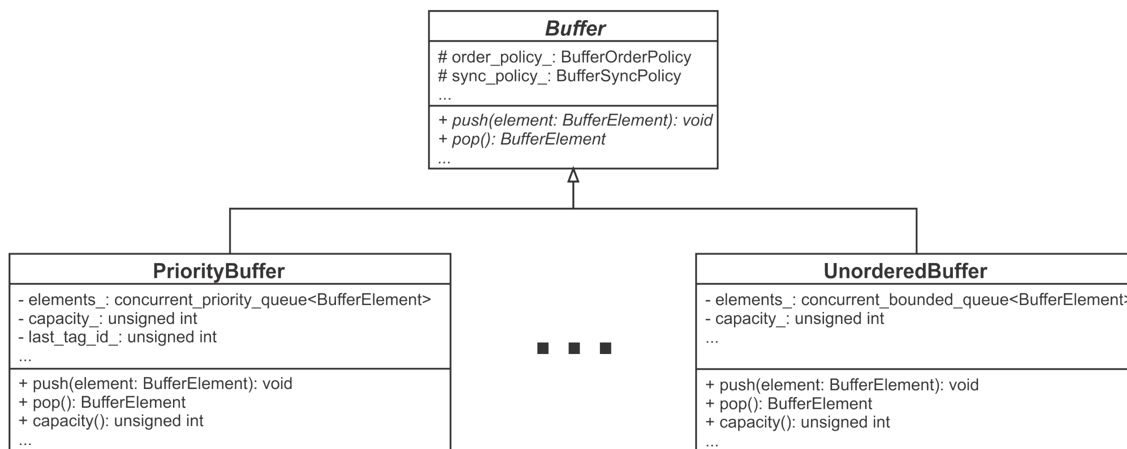


Figure 4.4: Buffer class diagram.

Data packets are encapsulated within a meta-object, thus allowing assignment of specific

information such as priority tags, creation dates and similar information whilst preserving the original data type. The *BufferElement* class encapsulates all such information. We expose two important buffer policy types addressing buffer element ordering and synchronization mechanisms for buffers. The element ordering refers to the order in which data packets are read from buffers when the *Pop()* method is invoked. In order to support stage execution apparatus, we want to ensure that the connected stages always get all the required data packets, hence we define the *Pop()* method to always perform a blocking call. Essentially, this will ensure that the stage *Execute()* method will wait until the correct element in the buffer is ready. The buffer synchronization policy is related to the type of data structures in use. Currently we support *bounded* and *unbounded* buffers, as well as *blocking* and *lock-free* queues. All policies are implemented strictly in the derived buffer classes. *Bounded* buffers provide the *capacity* attribute that describes the number of buffer elements that a buffer object may hold at a time. The *Push()* and *Pop()* methods are assumed to do all synchronization and waiting if required.

4.5.1 Buffer ordering policies

We distinguish between global and local buffer settings. While global settings are specified at pipeline level as an additional parameter of the *pipeline* construct, local settings may be adjusted at stage level. These local settings, as defined by the *stage* construct syntax, can override the global setting and allow different buffer configurations (e.g. change the capacity). The current implementation supports:

- *fifo* - First In, First Out policy which preserves the order. Data packets are returned in the same order they arrived.
- *priority* - buffers return data packets according to their *priority tag*. The priority tags can be automatically assigned or be a part of a data packet (e.g: frame numbers in the video compression applications).
- *unordered* - no order is preserved, packets may exit the pipeline/stage in an arbitrary order. This is not necessarily a disadvantage, since it may enable the layer to exploit more parallelism.

The current prototype implementation sets up the buffer policies upon the instantiation of the *Pipeline* object or indirectly via *PipelineManager*, while the buffer ordering polices are

configured when stages are linked (see section 4.3). A simple *enum* type is passed to signify the desired buffer policy (see listings 4.9 and 4.10 for available policies).

```
enum BufferOrderPolicy {
    BUFFER_ORDER_PRIORITY=0,
    BUFFER_ORDER_UNORDERED,
    BUFFER_ORDER_FIFO
};
```

Listing 4.9: Buffer ordering policies.

```
enum BufferSyncPolicy {
    BUFFER_SYNC_BOUNDED_TBB=0,
    BUFFER_SYNC_LOCKFREE_CHALMERS,
    BUFFER_SYNC_BOUNDED_STD,
    BUFFER_SYNC_UNBOUNDED_TBB
};
```

Listing 4.10: Buffer synchronization policies.

Finally, the number of buffer objects and their initial configuration is defined in the *Pipeline* class, which has been described in 4.3.

4.6 Pipeline Manager

The *PipelineManager* class facilitates dynamic management and (re-)configuration of pipelines. It comes with a set of methods that simplify access to the methods implemented in the *Pipeline* and *Runtime* classes. The coordination of pipeline stages is done in a decentralized manner using a local coordination strategy based on the stage callback mechanism (see section 4.7). Hence, *PipelineManager* is not used as a central controller for stage execution and coordination, but rather to control and manage the execution of (possibly many) pipelines within an application. This approach allows easier reconfiguration of the pipelines with the different set of parameters, such as replication factors, buffer sizes, etc.

The pipeline manager also serves as an interface for external tools (e.g., tools for automatic online tuning). Such tuning aspects are discussed more thoroughly in the “Conclusion and Future Work” section of the thesis. Additionally, the pipeline manager takes care of setting

up and managing the runtime system to be used for execution of pipelines. Currently, only the StarPU heterogeneous runtime system is supported.

```
// default constructor, the default settings are assumed
PipelineManager();

// Pipeline manager will use provided pipeline as the default one
PipelineManager( Pipeline *pipeline );

// specific runtime and buffer policy
PipelineManager(RuntimeType, BufferSyncPolicy);

// specific runtime configuration and buffer policy
PipelineManager(RuntimeConfiguration*, BufferSyncPolicy);

// specific runtime for preconfigured pipeline
PipelineManager(Pipeline *pipeline, RuntimeType );
```

Listing 4.11: Constructor methods of the *PipelineManager* class.

The implementation allows several ways of initializing the *PipelineManager*. Unless runtime and buffer synchronization policies are explicitly provided, the StarPU runtime and the *priority* buffer policy is used. The most important constructor methods are shown in Listing 4.11. Each constructor method will initialize the desired runtime with the provided or the default configuration. In addition, the *PipelineManager* class provides methods to execute pipelines in blocking or non-blocking way, to control its execution (e.g., pause/resume the execution), as well as the methods for querying statistical runtime data provided by the *Pipeline*, *Stage*, *Runtime* and *Buffer* classes.

Listing 4.12 demonstrates how the coordination layer is used to set up and start the simple four-stage linear pipeline shown in Listing 4.1. First, a *Pipeline* object is created and passed to the newly instantiated *PipelineManager* object. The transformation tool creates stage objects via the *Pipeline::CreateStage()* methods. Here, the desired position of a stage in the pipeline and optional stage names are provided in order to allow easier profiling. In the next step, the transformation tool generates stage parameters and stage ports based on the information queried from the PEPPHER component repository. Once all individual stages are configured, the *Pipeline::ConnectStages()* method is used to link the stage objects and dynamically create buffer objects according to the buffer creation policies and default settings. The actual stage connections are determined by the transformation system, based on the analysis of matching

component interface arguments in the high-level code representation.

The third argument of the *Pipeline::ConnectStages()* methods, allows different buffer orderings (see section 4.3). After all stages have been connected, their functionalities are assigned. The transformation system queries the available component implementations from the PEPPHER component repository and assigns the corresponding stage functionalities to the implementation variants wrapped in the *StageWrapper* function wrappers. Additionally, a control function (which relates to the original while loop condition) is assigned to the corresponding source stage. In the next step, the system sets up the stage replication factors and adjusts the buffer sizes if necessary. Finally, the pipeline manager is used to start the execution of the pipeline and to wait until the execution has completed.

During the execution, performance data, such as total execution time, number of buffer accesses, execution time of the individual stages, etc. is collected by the pipeline manager. Such performance data can be accessed after or during pipeline execution. The pipeline manager also manages reconfiguration of the pipeline if necessary (e.g., for online tuning or when re-running of the pipeline is required).

```
...
/* create new pipeline and pipeline manager objects */
Pipeline pipeline;
PipelineManager manager(&pipeline);

/* set-up the stages */
Stage *s1 = manager.CreateStage("s1", STAGE_TYPE_SOURCE);
Stage *s2 = manager.CreateStage("s2", STAGE_TYPE_MIDDLE);
Stage *s3 = manager.CreateStage("s3", STAGE_TYPE_MIDDLE);
Stage *s4 = manager.CreateStage("s4", STAGE_TYPE_SINK);

/* set-up the stage parameters */
s1->AddParameter((void *)&file,
                 VTT_CHAR, // type
                 s1_arg1_element_count, // number of elements
                 STAGE_PARAMETER_ACCESS_MODE_READ); // access mode
s1->AddParameter(VTT_CHAR,
                 s1_arg2_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_WRITE);
s2->AddParameter(VTT_CHAR,
                 s2_arg1_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_READ);
s2->AddParameter(VTT_CHAR,
                 s2_arg2_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_WRITE);
```

```
s3->AddParameter(VTT_CHAR,
                 s3_arg1_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_READWRITE);
s4->AddParameter(VTT_CHAR,
                 s4_arg1_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_READ);
s4->AddParameter((void *)&file,
                 VTT_CHAR,
                 s4_arg2_element_count,
                 STAGE_PARAMETER_ACCESS_MODE_READ)

 /* connect the stages */
 manager.ConnectStages(s1, s2);
 manager.ConnectStages(s2, s3);
 manager.ConnectStages(s3, s4, BUFFER_ORDER_PRIORITY);
/* assign stage functionalities and control functions */
s1->add_functionality(StageWrapper1, STAGE_IMPLEMENTATION_CPU);
s1->SetControlFunction(WhileLoopWrapper);
s2->add_functionality(StageWrapper2, STAGE_IMPLEMENTATION_CPU);
s3->add_functionality(StageWrapper3, STAGE_IMPLEMENTATION_CPU);
/* add a CUDA enabled GPU implementation variant for s3 stage */
s3->add_functionality(StageWrapper3_GPU, STAGE_IMPLEMENTATION_CUDA);
s4->add_functionality(StageWrapper4, STAGE_IMPLEMENTATION_CPU);

/* Adjust replication factors */
s3->set_replication_factor( rf ); // stage 3 replication factor

 /* create new runtime configuration */
 RuntimeConfiguration rconf(STARPU_RUNTIME,
                            __vtt_NCPUS, // NCPUS
                            __vtt_NGPUS, // NGPUS
                            __vtt_SCHEDULING_POLICY); // SCHEDULER

/* instruct PipelineManager that the pipeline will use the StarPU runtime */
manager.init(&pipeline, &runtime_conf);

/* execution of the pipeline as a whole or in segments */
if ( segment_size <= 0 ) {
  manager.RunPipeline();
} else {
  while ( !manager.pipeline()->terminated() )
    manager.RunPipeline( segment_size );
}
...
```

Listing 4.12: The code that utilizes the coordination layer to execute the four-stage pipeline described in Listing 4.1.

4.7 Pipeline Execution and Coordination

While the coordination layer uses a certain number of threads to orchestrate parallel pipeline execution, the actual number of threads used by an application that utilizes the layer may vary depending on various factors (e.g., number of pipeline stages or runtime system in use). In general, the layer spawns one thread for each pipeline stage or stage replica. These stage threads live until the pipeline is terminated or replication factors are decreased, in which case the threads executing respective replicated objects are terminated. Moreover, one additional management thread per pipeline is required to support asynchronous execution and control of pipelines.

Classes relevant to execution of the pipeline are the *Pipeline*, *Stage* and *PipelineManager* classes. Creation, management and synchronization of threads is built into the *Pipeline* and *Stage* classes. In this context, the *Pipeline* class provides methods to control execution, configuration and synchronization of the pipeline as a whole. In the current prototype, pipelines may be executed until all data packets are processed, or in segments of a variable size.

Stage execution, coordination and synchronization is implemented in the *Stage::Execute()* and *Stage::Callback(...)* methods of the *Stage* class. Every stage instance is executed in an asynchronous fashion, via the *Stage::ExecuteAsync()* method. The *PipelineManager* class is used to provide control over multiple pipelines, as well as the methods for selection of runtime system and the wrappers for handling the structure of the pipelines and controlling its execution (replication factors, buffer sizes, queuing policies, starting and pausing the pipeline, executing in segments, etc.).

Instead of running and scheduling tasks directly, the coordination layer submits the tasks for execution to the runtime system. Currently, the supported StarPU heterogeneous runtime system provides a callback mechanism for scheduled tasks [23]. The callback behavior is defined via the *callback()* function that is called upon the termination of a scheduled task.

Overall execution of the pipeline is controlled by the methods implemented in the *Pipeline* class:

- *ExecuteAsync()* - asynchronously executes the pipeline. This method will spawn/awake stage threads and start/resume the pipeline execution
- *Wait()* - explicit wait command. This method will halt the execution until a pipline

segment finished executing, or the pipeline is paused

- *Pause()* - pauses the execution of the pipeline
- *Resume()* - resumes the execution of the pipeline
- *Reconfigure()* - allows reconfiguration of the pipeline if any of the structural parameters are changed. For example, reconfiguration after replication factors have been modified will ensure that respective stage replicas and corresponding threads are created/terminated.

All pipeline and stage properties can be changed dynamically during the execution. However, some changes may require the pipeline to be paused, reconfigured and resumed.

An overview of the execution process is shown in Figure 4.5. After the pipeline, stage and buffer objects are created, the pipeline manager starts the pipeline by invoking the *PipelineManager::RunPipeline()* method. This method is a wrapper for the *Pipeline::ExecuteAsync()* method that calls *Stage::ExecuteAsync()* for each stage object in the pipeline. The *ExecuteAsync()* method of the *Stage* class creates one thread for each stage (or stage replica) and calls the *Stage::Execute()* method.

We realized a stage-local coordination strategy, rather than handling the coordination by a global coordinator. Once the threads have been spawned, the stages continue executing on their own by utilizing the callback mechanism. In the general case, the *Stage::Execute()* method is calling the *Pop()* method for all input buffers and continues once a buffer element (data packet) is received. The wait mechanism is implemented in the *Buffer* class where *Buffer::Pop* is a blocking call. Since the StarPU runtime supports management of data transfers, each task if first configured with the appropriate data handles, and then created and posted to the runtime for execution. The creation, configuration and scheduling of the tasks is accomplished in a separate step by the *Runtime* class. The *Runtime* class is an abstract class, so concrete classes might implement their own task creation, configuration and scheduling methods.

In the case of StarPU, tasks are described by *codelets* ("a unified offloadable task abstraction" [23]) - the StarPU data structure that describes a computational kernel that may feature implementations for multiple architecturally different execution units, such as CPU, CUDA or OpenCL devices. Stage functionalities and information on stage ports is passed to the

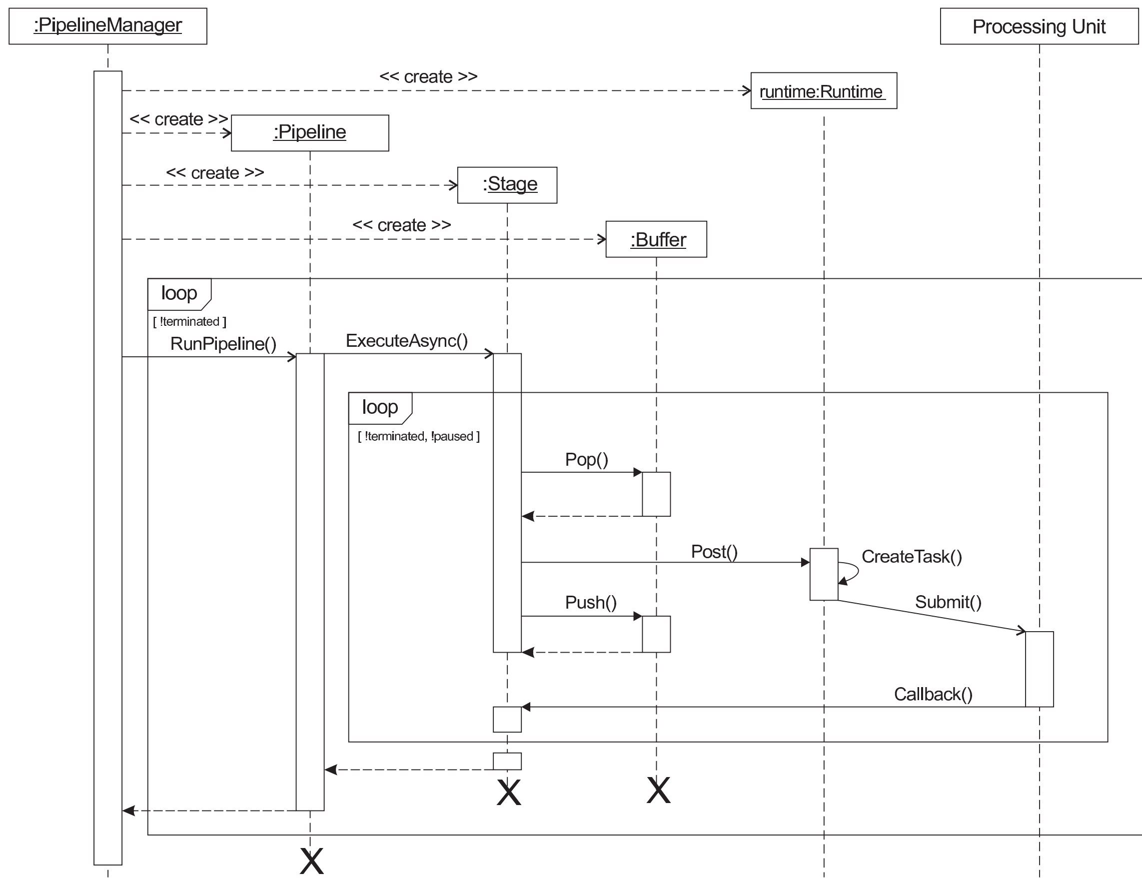


Figure 4.5: Execution model sequence diagram. This diagram describes the execution of the pipelines. *PipelineManager* creates the pipeline, stages and buffer objects and initiates the execution of the stage threads. The *Stage::Execute()* method will post the tasks for execution until the termination criterion is reached.

codelet structures, providing the runtime with necessary information for making scheduling decisions. Additionally, tasks use the appropriate data handles that are provided by buffers and contained in the *BufferElement* object (see 4.9.1). Since this stage structure is not likely to change over the course of a single application run, this approach may reuse the same *codelet* structure for all tasks produced by one stage. In the next step, the runtime selects the suitable implementation variant and the task is then scheduled for the execution on a worker (e.g. computation resource [23]). Meanwhile, the stage pushes the task handles into output buffer objects. This way, the task dependencies are correctly defined for the tasks spawned by the successor stages as well.

Once the task is posted to the runtime system and the *Buffer::Push()* method has completed, the *Stage::Execute()* method waits until the task has finished and the runtime *callback()* func-

tion is invoked. This will trigger the *Stage::Callback()* function that will perform termination checks and call the *Stage::Execute()* again. This process is repeated until the termination criteria is met. Essentially, the pipeline and stage termination is triggered when there no more data to process. The stage execution mechanisms are also supported by the methods that allow execution to be paused or resumed according to the state of the pipeline, as well as the mechanisms for collecting statistical execution data for each stage (e.g., execution time, input or buffer wait times, number of executions or similar).

```
int pipeline_iterations = 100;
  manager.RunPipeline( pipeline_iterations );
```

Listing 4.13: Executing a pipeline in segments

As opposed to the execution of the whole pipeline at once, the pipeline may also be executed for a predefined number of iterations (see Listing 4.13) only. In this case, the pipeline *Pause()* method will be called as soon as all stages perform specified number of iterations. Pausing the pipeline implies putting all stage threads in the wait state, until the pipeline *Resume()* is called. Therefore, the pipeline pause is usually a costly operation.

4.8 Runtime and Scheduling

4.8.1 StarPU Runtime

StarPU is a runtime system for scheduling a DAG of tasks onto heterogeneous set of processing units in portable and efficient way [37, 23]. Several aspects of the StarPU runtime are making it especially appropriate choice for our framework. Above all, it supports tasks with multiple implementations, targeting different execution units of the available hardware. Next, data transfers can be handled automatically, thus avoiding unnecessary data copying and keeping data closer to the execution units. Additionally, estimation of the data transfer costs allows the runtime to optimize the transfers, perform data pre-fetching and utilize different scheduling alternatives [37].

StarPU ships with a set of predefined scheduling strategies that address efficient scheduling of tasks to the execution units of heterogeneous system. By default, StarPU uses the well-known *eager* scheduling policy. However, particularly interesting schedulers for this work are those that take historic performance data into account. Heterogeneous earliest finish time (HEFT), also known as deque model data aware (DMDA) scheduler, schedules tasks to workers taking into account the current system load, available component implementation variants, and historical execution profiles, with the goal of minimizing overall execution time by favoring implementation variants with the lowest expected execution time.

4.8.2 Runtime Class

In the coordination layer, the *Runtime* class is an abstract class that defines the interfaces for all runtime systems that can be used by the coordination layer. The *StarPURuntime* class is accordingly a concrete implementation of the *Runtime* class that allows StarPU to be used with the layer.

There are two important aspects of the StarPU runtime that we have tried to exploit in order to allow efficient interplay with the coordination layer. The first aspect relates to the already mentioned callback mechanism (see section 4.7) and the StarPU execution model in general, where the *codelets* are used as an abstraction of a computational task that can be executed on a processing element of a heterogeneous system (e.g., CPU core, GPU, etc..). This approach is tightly coupled with the PEPPHER component model, so that component implementation variants correspond to the StarPU *codelet* implementations for different architectures, coded using different programming languages (e.g., CUDA, OpenCL, etc.). The second aspect refers to the ability of the StarPU runtime to manage data transfers automatically [23]. In fact, StarPU ships with a powerful data management system that aims to optimize data transfers. Thus, the coordination layer shifts the actual data management to the runtime.

Usually, initialization and termination of the runtime system is the responsibility of the *PipelineManager*, however this may be overridden by setting up and reconfiguring the runtime directly with the methods provided by the *Runtime* class. Figure 4.6 shows important methods of the runtime class.

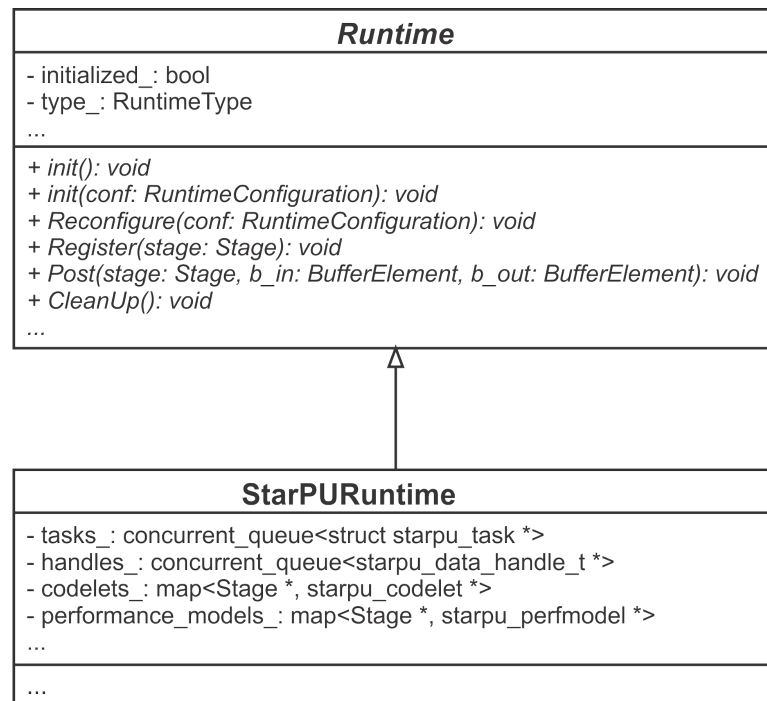


Figure 4.6: *Runtime* and *StarPURuntime* classes.

4.9 Other classes

This section briefly covers relevant methods and properties of several other classes in the coordination layer.

4.9.1 BufferElement

The *BufferElement* class encapsulates all the information related to the data packets used by buffers. Essentially, this includes runtime handles, references to sender and receiver stages, data packet tags (priority tags), and actual pointers to the data.

4.9.2 StageParameter

The *StageParameter* class considers various configurations of the stage parameters, which correspond to function arguments as specified in the PEPPHER component repository. Hence, a stage parameter object includes the information about data type, element count, access

modes and flow direction (i.e.: in, out, inout). The class provides several constructor methods that allow easier setup of stage parameters (see Listing 4.12).

4.9.3 StagePort

A stage port corresponds to a set of stage parameters. The implementation of the *StagePort* class follows this approach by aggregating all parameters of the ports in the *std::vector* container in the following way:

```
...
std::vector<StageParameter *> stage_parameters_;
...
```

Listing 4.14: *StagePorts* are collections of *StageParameters*.

4.10 Performance Measurement Facilities

The coordination layer interacts with the runtime system that performs dynamic scheduling of tasks. Often, a runtime system employs different scheduling policies and makes execution decisions on its own. Hence, measuring performance is not a straightforward and easy task. In the following we describe the performance metric supported by the coordination layer.

4.10.1 Execution and Congestion Time

The layer measures overall pipeline execution time, as well as the execution time of every stage in the pipeline. The pipeline execution time corresponds to a time interval between pipeline start and its termination, not counting the time pipeline has spent in paused state.

Measuring the stage execution time is slightly more complex because the actual execution is delegated to the runtime system. The time needed for processing the same data packet may be different for each stage of the pipeline. Different component implementations may perform differently, causing the pipeline to be imbalanced, and input and output buffers wait

times to increase unevenly. Therefore, we also measure buffer congestion in terms of input and output buffer wait times.

Furthermore, as all stage threads are spawned at approximately the same time, this implies that all stage threads will wait for data as soon as the pipeline is started. In the case of replicated stages individual execution times may overlap , as shown in Figure 4.7.

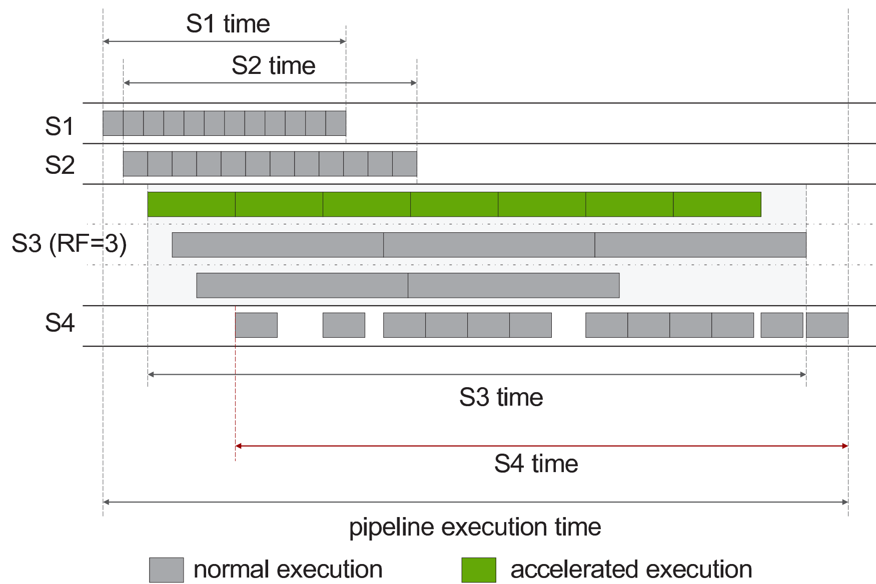


Figure 4.7: Performance measurements for pipelines with replicated stages. Stage *S3* has been replicated by the factor of 3.

We measure the stage execution time as a cumulative execution time of all stage invocations, excluding the overlapping time. Since stage tasks are executed by the runtime system, this time is also referred to as stage runtime processing time.

4.10.2 Throghput

We define the *pipeline throughput* as the measure of the number of data items that pipeline can process per time unit. For example, if we process 100 data packets in 20 seconds, then measured pipeline throughput (*TP*) would be:

$$TP_{pipeline} = \frac{100}{20} = 5$$

The stage throughput is measured in the similar way and although it delivers some quantitative measure for stages, there are a few caveats. Let's assume a linear pipeline, and that we want to use the stage throughput to decide which stage to replicate. The most time consuming stage is usually referred to as the *limiter stage*. When measured in a way described above, the limiter stage of a pipeline also limits the throughput of all successor stages. Hence, it is giving false impression that successor stages have the same performance demands as the limiter stage. The effect can be observed regardless of the stage replication factor (see figures 4.7 and 4.8).

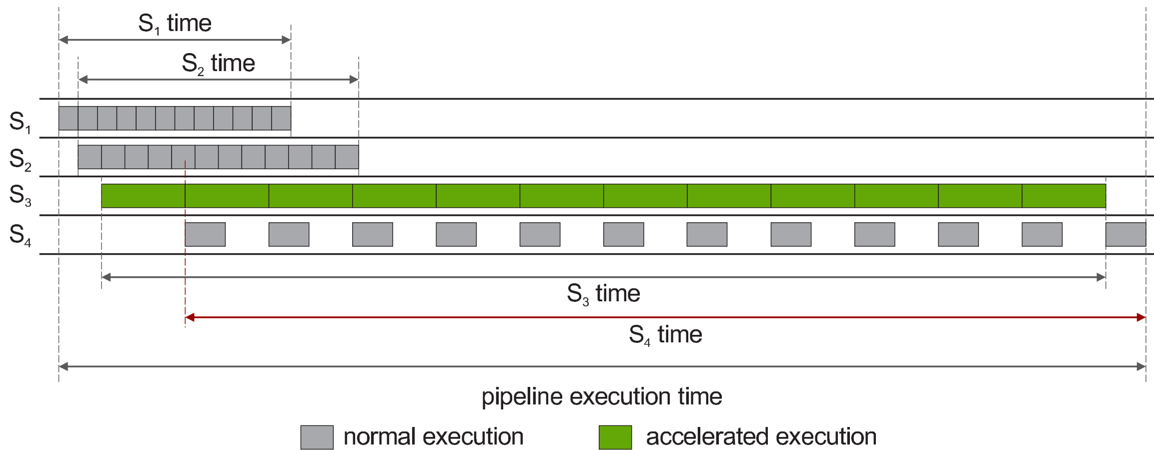


Figure 4.8: Performance measurements for pipelines with non-replicated stages.

One way to address this issue is to take into account the time a stage waiting for data to arrive. In fact, the actual limiter stage will have a shorter wait time when compared to the faster successor stage.

The coordination layer comes with a set of methods that enable querying of the described performance data. While the *Pipeline* class provides *Pipeline::ExecutionTime()* method for measured execution time of the whole pipeline or of the last segment, the *Stage* class provides somewhat richer set of methods. *Stage::InputWaitTime()* returns the time used by the stage while waiting for the required data packets to arrive. Similarly, *Stage::OutputWaitTime()* gives the time used by the stage object while waiting for the processed data packets to leave congested stage output buffers.

The layer provides two different execution times for each stage. While *Stage::ExecutionTime()* returns the execution time of the component implementation variants (e.g., CPU/GPU computational kernel), *Stage::RuntimeProcessingTime()* return the total time needed by the runtime system to process selected component implementation variant. These two execution

times may differ since the value returned by the *Stage::RuntimeProcessingTime()* method measures the time interval that starts from the actual submission of the task to the runtime system and ends when the task has finished executing.

5 Source-To-Source Transformation

Automatic transformation of high-level codes with PEPPHER parallel pipeline patterns to a code that utilizes the pipeline coordination layer is supported by a source-to-source compiler (the transformation tool). In this chapter, we give a short overview of the ROSE compiler framework, which is used to build the transformation tool. Afterwards, we provide a brief description of the transformation process followed by an example that outlines the transformation of a three-stage image processing pipeline.

5.1 Rose Framework

ROSE is an open-source compiler infrastructure for building source-to-source program transformation and analysis tools [38]. It supports the development of translators/compilers that perform automated analysis, transformation, instrumentation, data extraction or similar actions on source code. ROSE ships with a large set of customizable tools, it supports parsing, common forms of compiler analysis, common transformations and code generations. Such tools relieve users of the need to re-implement some of the common compiler features, speeding up the development time. ROSE is an ongoing project, pushed forward by a group at Lawrence Livermore National Laboratory (LLNL) and the number of supported features it delivers constantly grows.

As modern compilers tend to be very complex, at the conceptual level their structure is often organized in phases. In general, these phases can be classified into three major groups: *frontend*, *midend* and *backend*. The frontend usually takes care about parsing source code and converting it into a structured form, imposing the grammar, performing lexical analysis, syntax analysis, type checking, and creating an intermediate representation (IR). The midend deals with manipulation of the intermediate representation, and may include various optimizations and transformations. Finally, the backend handles code synthesis, assuming generation of the code in target language, perhaps assembly, linking and register allocation [39].

Such recognizable structure of modern compilers can also be found in ROSE. It uses multiple frontends, namely, Edison Design Group (EDG) frontend for parsing C, C99, UPC, C++ codes and Open Fortran Parser (developed at Los Alamos National Laboratory) for parsing Fortran F2003/F77/F90/F95 codes. The EDG frontend is proprietary, fully commercial C/C99/UPC/C++ language parser, hence only its binaries are provided in ROSE. For the intermediate representation ROSE internally uses object-oriented SAGE III IR, which is an extension of SAGE II/SAGE++ IR and makes sure that all of information from the original source code is preserved including C preprocessing structure, source comments, source position information, C++ template information, etc. Additionally, ROSE adds many features to SAGE III, such as loop analysis and optimizations, abstract syntax tree (AST) rewrite, query and visualization support mechanisms, C++ template support, etc. A set of easy to use interface functions makes it particularly useful for writing prototype language extensions in a productive way and thus not imposing a requirement for deep compiler experience for a user.

In a typical usage scenario ROSE operates in such a way that a source code written in C, C++ or FORTRAN is read and an object-oriented AST is constructed. Essentially, AST forms a graph representing the structure of the source code and it exposes interface functions enabling traversal, analysis and manipulation of the tree, including support for control flow, data flow, class hierarchies, data and system dependence analysis, etc. After the transformation of the code is finished, the backend can perform generation of targeted code (unparse the AST), eventually compiling it with support of vendor compilers.

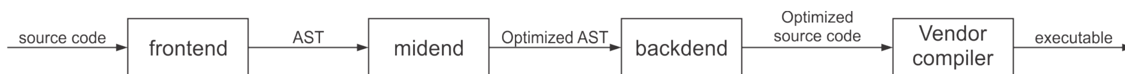


Figure 5.1: High-level overview of possible source-to-source transformation process within the tools built with ROSE.

Many factors supported the decision to use ROSE as a compiler framework. This includes the support for source-to-source transformation, choice of supported frontends, easy to use interfaces for AST manipulation and transformation, as well as the increasing support for parallel applications, ongoing development and open-source license and availability.

Nevertheless, ROSE represents C/C++ pragma directives with plain strings and no special processing is performed. Therefore for the purpose of our source-to-source transformation compiler, simple C/C++ pragma directive parser has been developed.

5.2 Transformation Process

In this section we will go through the transformation process more extensively. Currently, the transformation tool aims to support transformation of C/C++ code with the annotations as defined in the context of the PEPPHER project. The transformation tool is based on ROSE, which implies that source code is parsed with the EDG frontend and the only limitations in that regard are those imposed by EDG itself.

First, the transformation tool performs the usual frontend processing of the original source code with PEPPHER pipeline annotations and builds the AST. In order to extract user-provided information, the AST is traversed to find all while loops annotated with the PEPPHER pipeline pragma directives. If such loops are found, the pragma directives, which are represented with a strings, are passed to the pragma parser for further processing. The pragma parser extracts the user-provided information from the pragma strings according to the PEPPHER pipeline syntax (see Chapter 3.3), and attaches it to the corresponding while loops in the form of simple C++ structs.

Next, the tool builds the output code according to the code structure and the acquired user-information. For each properly annotated while loop the tool will create and initialize the *Pipeline* object, and register it to the *PipelineManager* object, which is then created together with the first *Pipeline* object. The output code (see Listing 5.1) will feature one *PipelineManager* object and it will use the default or user-provided buffers and the STARPU runtime system. All detected pipelines will be registered with the pipeline manager.

```
Pipeline pipeline;
PipelineManager manager(STARPU_RUNTIME, &pipeline);
```

Listing 5.1: Instantiation of the *Pipeline* and *PipelineManager* objects. Once the pipeline is created it is registered with the pipeline manager.

Afterwards, the AST of each loop body is traversed and component calls are detected and related to the stages of the corresponding pipeline. In order to create stage objects the transformation tool combines user-provided information (e.g., modified replication factors, buffers, etc.) with the corresponding information from the PEPPHER component repository (e.g. interface parameter access modes, component implementation variants) and the information on the data dependencies between arguments passed to component interfaces. The data de-

pendencies are described by a DAG, which is used to identify stage ports and connect stages objects. The output code will feature stage wrappers for actual calls to component implementations, the wrappers for the while loop condition, and stage objects configured with user-provided information (see Listing 5.2).

```
...
void StageWrapper1 (void *descr[], void *args) {
  // set up the ptrs with the help of runtime system (StarPU)
  char *__vtt_arg0 = (char *) STARPU_VECTOR_GET_PTR(descr[0]);
  ...
  component_1(__vtt_arg0, __vtt_arg0_size, ...);  // function call
}
...
bool WhileLoopWrapper() {
    ...
}
...
Stage *s = manager.pipeline(0)->CreateStage("s", STAGE_TYPE_SOURCE);

s->AddParameter(VTT_CHAR, size, STAGE_PARAMETER_ACCESS_MODE_WRITE);
...
s->add_functionality(StageWrapper1, STAGE_IMPLEMENTATION_CPU);
s->add_functionality(StageWrapper2, STAGE_IMPLEMENTATION_GPU);
s->SetControlFunction(WhileLoopWrapper);
...
s->set_replication_factor(4); // user-provided
...
```

Listing 5.2: Wrapping component calls and while loop condition into stage wrapper functions and creation of stage objects

Finally, the transformation tool generates calls to kick-start the execution of the pipeline (see Listing 5.3).

```
if ( segment_size <= 0 ) {
  manager.RunPipeline(&pipeline);
} else {
  //Execution in segments:
  while ( !manager.pipeline()->terminated() ) {
  manager.RunPipeline( &pipeline, segment_size );
}
```

Listing 5.3: Starting up the pipeline and waiting for the execution to finish.

5.3 Transformation Example

Let us closely examine this process by an example. For instance, let us assume we want to process a stream of images. Looking at the problem from a higher perspective, image processing process consists of a three stages - loading, processing and saving of each image in the stream. Hence, we can form a simple linear three stage pipeline. Listing 5.4 shows a sequential while loop that processes a stream of data. Additionally, it shows annotations used by the transformation system.

```
#pragma pph pipeline
while ( inputstream >> file ) {
  load ( file, image ); // loads the image from stream
  #pragma pph stage replication (4) // replicate stage 4 times
  process ( image ); // process the image
  #pragma pph stage buffer ( PRIORITY, 32 )
  save ( file, image ); // save the processed image
}
```

Listing 5.4: Transformation example: Simple image processing pipeline.

In the first step, the transformation tool will process the while loop, create the AST and enrich it with information parsed from pragma annotations. Next, each component call will be analyzed in order to correctly create corresponding stage objects. The component interfaces will be used to query the PEPPHER component repository for the concrete component implementation variants. For each component implementation variant, the transformation tool generates a stage wrapper and associates it with the corresponding pipeline stage as a stage functionality (see Listing 5.5).

```
/* Stage function wrappers */
void StageWrapper1 (void *descr[], void *args) {
  char *__vtt_arg0 = (char *) STARPU_VECTOR_GET_PTR(descr[0]);
  unsigned int __vtt_arg0_size = STARPU_VECTOR_GET_NX(descr[0]);
  uchar *__vtt_arg1 = (uchar *) STARPU_VECTOR_GET_PTR(descr[1]);
  unsigned int __vtt_arg1_size = STARPU_VECTOR_GET_NX(descr[1]);

  load(__vtt_arg0, __vtt_arg0_size, __vtt_arg1, __vtt_arg1_size);
}

void StageWrapper2 (void *descr[], void *args) {
  uchar *__vtt_arg0 = (uchar *) STARPU_VECTOR_GET_PTR(descr[0]);
  unsigned int __vtt_arg0_size = STARPU_VECTOR_GET_NX(descr[0]);
```

```
    process(__vtt_temp0, __vtt_temp0_size);
  }

  #ifdef vtt_GPU_CAPABLE
  void StageWrapper2_GPU (void *descr[], void *args) {
    uchar *__vtt_arg0 = (uchar *) STARPU_VECTOR_GET_PTR(descr[0]);
    unsigned int __vtt_arg0_size = STARPU_VECTOR_GET_NX(descr[0]);

    process(__vtt_arg0, __vtt_arg0_size);
  }
  #endif

  void StageWrapper4 (void *descr[], void *args) {
    char *__vtt_arg0 = (char *) STARPU_VECTOR_GET_PTR(descr[1]);
    unsigned int __vtt_arg0_size = STARPU_VECTOR_GET_NX(descr[1]);
    uchar *__vtt_arg1 = (uchar *) STARPU_VECTOR_GET_PTR(descr[0]);
    unsigned int __vtt_arg1_size = STARPU_VECTOR_GET_NX(descr[0]);

    save(__vtt_arg0, __vtt_arg0_size, __vtt_arg1, __vtt_arg1_size);
  }
```

Listing 5.5: Transformation example: Wrapping the component implementation variants.

The while loop condition is decoupled and wrapped in the function that returns a boolean value (see Listing 5.6). This wrapper function will be assigned as a control function to source stage(s) of the pipeline.

```
/* while loop wrapper */
bool WhileLoopWrapper() {
  if ( inputstream >> file ) {
    // do not send termination signal
    return true;
  } else {
    // set termination trigger
    return false;
  }
}
```

Listing 5.6: Transformation example: Wrapping the while loop condition.

The resulting output code will feature insertion of the code on two levels: in the global scope and in the scope of the original while loop. In the global scope, before the *main* function the tools inserts new includes for the coordination layer, stage wrapper functions and while

loop wrapper functions. In the scope of the original while loop, *while loop* is removed and calls to the coordination layer are inserted.

In order to use the coordination layer, the transformation tool will first create and initialize the *Pipeline* object, which is then used to create stage objects. Additionally, the *PipelineManager* object is created and configured to be use the StarPU runtime system with the newly created *Pipeline* object (see Listing 5.7).

```
int main(int argc, char *argv[]) {
...
  Pipeline pipeline;
  PipelineManager manager(STARPU_RUNTIME, &pipeline);

  Stage *s1 = pipeline.CreateStage("__vtt_example_stage1", STAGE_TYPE_SOURCE);
  Stage *s2 = pipeline.CreateStage("__vtt_example_stage2", STAGE_TYPE_MIDDLE);
  Stage *s3 = pipeline.CreateStage("__vtt_example_stage3", STAGE_TYPE_SINK);
...
}
```

Listing 5.7: Transformation example: Creating and initializing *Pipeline*, *PipelineManager* and *Stage* objects.

Afterwards, the transformation tool uses the information from the DAG analysis of the component interface arguments to describe the stage parameters and connect the stages (see listings 5.8 and 5.9).

```
...
  s1->AddParameter((void *)&file, VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);
  s1->AddParameter(VTT_UCHAR, img_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s2->AddParameter(VTT_UCHAR, img_size, STAGE_PARAMETER_ACCESS_MODE_READ);
  s2->AddParameter(VTT_UCHAR, out_img_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s3->AddParameter(VTT_UCHAR, out_img_size, STAGE_PARAMETER_ACCESS_MODE_READ);
  s3->AddParameter((void *)&file, VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);
...
```

Listing 5.8: Transformation example: Setting up the stage parameters.

Stages are connected through the *Pipeline::ConnectStages(...)* methods. The usage of the *buffer* construct in high-level codes will cause different buffers to be inserted between the

stages. Listing 5.9 shows that stages *s1* and *s2* have been connected via default buffers, while between *s2* and *s3* a *PriorityBuffer* has been inserted.

```
...
  pipeline->ConnectStages(s1, s2);
  pipeline->ConnectStages(s2, s3, BUFFER_ORDER_PRIORITY);
...
```

Listing 5.9: Transformation example: Connecting the stages via *Pipeline*.

Listing 5.10 shows how stages functionalities are defined. The number of stage functionalities is determined by the number of available component implementation variants. In Listing 5.10 stage *s3* features CPU and GPU implementation variant.

```
...
  s1->add_functionality(StageWrapper1, STAGE_IMPLEMENTATION_CPU);
  s1->SetControlFunction(WhileLoopWrapper);

  s2->add_functionality(StageWrapper2, STAGE_IMPLEMENTATION_CPU);

  s3->add_functionality(StageWrapper3, STAGE_IMPLEMENTATION_CPU);
  #ifdef vtt_GPU_CAPABLE
  s3->add_functionality(StageWrapper3_GPU, STAGE_IMPLEMENTATION_CUDA);
  #endif
  s4->add_functionality(StageWrapper4, STAGE_IMPLEMENTATION_CPU);
...
```

Listing 5.10: Transformation example: Setting up stage functionalities and control functions.

Listing 5.11 demonstrates the configuration of the stage replication factor and input buffer size.

```
...
  s3->set_replication_factor(4);
  s4->input_buffer()->set_capacity(32);
...
```

Listing 5.11: Transformation example: Adjusting stage replication factors and buffer sizes.

Finally, the *PipelineManager* object is used to start the execution and to wait for the execution to finish (see Listing 5.12).

```
...
    if ( segment_size <= 0 ) {
        manager.RunPipeline();
    } else {
        //Execution in segments:
        while ( ! manager.pipeline()->terminated() ) {
            manager.RunPipeline( segment_size );
        }
...
```

Listing 5.12: Transformation example: Starting up the pipeline and waiting for the execution to finish.

6 Experiments and Evaluation

The primary goal of this chapter is to confirm the viability of our approach by evaluating the performance of two different applications that utilize our framework. Particularly interesting are applications processing streams of data such as digital signal processing, encryption, image processing and graphics in general.

We have performed an evaluation of the framework using two applications from the domains of computer vision and bioinformatics - both featuring component implementation variants for CPU cores and GPUs. These implementation variants have been taken from publicly available open-source libraries and adapted to support the PEPPHER component model. The results achieved with our approach are compared against baseline implementations that consider sequential execution and parallel execution on CPU cores and/or graphic cards as supported by the original libraries. Additionally, for each application we demonstrate an annotated source code and compare it to the transformed code that utilizes the coordination layer for pipelining.

First, we present three different heterogeneous systems we used for the performance evaluation. Afterwards, we dedicate a separate section to each experiment that has been conducted. Finally, we end this chapter with a short summary.

6.1 Target Hardware

Performance measurements have been conducted on three different heterogeneous systems hosted at the University of Vienna.

The first system, named "NADIA", houses two Intel X5550 processors, members of NEHALEM-EP architecture [40] operating at frequency of 2.66 GHz. L1 and L2 cache sizes follow the standard NEHALEM micro architecture design, providing 32KB data and 32KB instruction L1 cache and 256KB of L2 cache per core. Each processor features four cores supporting Intel Hyper-Threading technology and 8MB of L3 cache. Additionally, this system includes two NVidia graphic cards - NVidia GTX 285 based on NVidia "Tesla" microarchitecture featuring 240 CUDA cores and 1GB GDDR3 device memory, and NVidia GTX 480 representative of "Fermi" [41] architecture and comprising 480 CUDA cores and 1536 MB GDDR5 device memory.

The second system is a single node of the "PHIA" cluster. This node houses two eight-core Intel Xeon E5-2650 CPU and four NVIDIA Tesla K20 M-class GPUs. Intel Xeon E5-2650 [42] is a member of a newer Intel Sandy Bridge processor family normally operating at 2.0 GHz, but with an ability to increase the frequency up to 2.8GHz via Intel Turbo Boost technology. L1 and L2 cache sizes follow the standard Sandy Bridge micro architecture design, featuring 64KB of L1 cache and 265KB of L2 cache per core. Having 20 MB of L3 cache is a significant improvement compared to the processor in NADIA system. In total there are 16 physical cores and 128GB of RAM available to the end user. NVIDIA Tesla K20 M-class cards are representatives of NVidia Kepler GK110 microarchitecture [43], featuring 2496 CUDA cores and 5 GB GDDR5 device memory. The system runs a 64-bit version of Red Hat Enterprise Linux 6.

The third machine is also a heterogeneous system. A single node of "CORA" system consists of two Intel Xeon X5550 processors running at 2.66 GHz. Additionally, this system includes three NVidia graphic cards - two NVIDIA Tesla C2050 GPUs and one NVidia Tesla C1060 GPU. The NVIDIA Tesla C2050 GPUs are representatives of "Fermi" [41] architecture. Each comprising 448 Streaming-cores and hosting 3 Gigabytes GDDR5 device memory. NVidia Tesla C1060 is a slightly older model, comprising 240 streaming cores and 4 GB of GDDR3 memory. The system runs a 64bit version of Red Hat Enterprise Linux 5.

6.2 Face Detection Application

Evolving hardware architectures with an increasing amount of computational power for cheaper prices, along with the wider availability of the input/output devices (camera, sensors, etc.) have enabled advances in many areas of computer science. In the field of computer vision this translates to use of emerging algorithms for image and video processing in a real time. For the purpose of this work we will use the definition of computer vision as "the transformation of data from a still or video camera into either a decision or a new representation" from [32].

The Open Source Computer Vision (OpenCV) is an open-source, multi-platform computer vision and machine-learning library [32]. Although the library is written in C/C++, it features interfaces for Python and Java and provides support for multi-core processors as well as for CUDA/OpenCL enabled GPUs. Moreover, OpenCV can take advantage of various libraries such as Intel Integrated Performance Primitives (IPP) [44] and Intel Thread Building Blocks (TBB) [45, 46].

6.2.1 Description

Based on the PEPPHER high-level coordination constructs, we have implemented an image processing pipeline for detecting human faces on a stream of images. This example makes use of PEPPHER multi-architectural components with implementation variants for CPU and GPUs re-engineered from the OpenCV library. Essentially, the algorithm processes an input data stream, which consists of a series of video frames or a set of images, and annotates each image in the data set with rectangles signifying the position of detected human faces.

We split the whole process into four pipeline stages: *ReadImage*, *ConvertAndResize*, *DetectFaces* and *WriteImage* (see Listing 6.1).

The first stage in the pipeline is the *ReadImage* stage, which simply loads an image from the input data set (usually a set of files or a video stream). In order to prepare the image for face detection and improve the efficiency of the main algorithm, the *ConvertAndResize* stage takes care of image conversion from color to grayscale, and possible re-sizing. The computationally most intensive part of the application is contained in the *DetectFaces* stage, which is relatively slow compared to other stages. It uses the face detection algorithm based on the

Viola-Jones [47] approach. Since this detection algorithm relies on trained object models, the detection stage takes care of loading pre-trained object models as well. In our example, we use models that ship with OpenCV library. Finally, the *WriteImage* stage annotates images with rectangles and writes them to disk.

```
...
int main(int argc, char *argv[]) {
...
    while ( *inputstream >> filename ) {
        ReadImage(filename, name_size, img_data, img_size);
        ConvertAndResize(img_data, img_size, out_img_data, out_img_size);
        DetectFaces(out_img_data, out_img_size);
        WriteImage(filename, name_size, out_img_data, out_img_size);
    }
...
}
```

Listing 6.1: Face Detection: Application utilizes PEPPHER components (re-engineered from OpenCV) to detect human faces on a series of images.

6.2.2 Annotated Code

Using the support for pipeline patterns we have annotated a sequential face detection source code with the pipeline directives (see Section 3.3).

```
...
    #pragma pph pipeline
    while ( *inputstream >> filename ) {
        ReadImage(filename, name_size, img_data, img_size);
        ConvertAndResize(img_data, img_size, out_img_data, out_img_size);
        #pragma pph stage replication(rf)
        DetectFaces(out_img_data, out_img_size);
        WriteImage(filename, name_size, out_img_data, out_img_size);
    }
...
```

Listing 6.2: Face Detection: Annotated compute intensive part of the example. The while loop is marked to be transformed into a parallel pipeline with the PEPPHER pipeline directives.

The while loop with PEPPHER components (shown in Listing 6.2) is a compute intensive part of the Face Detection example. The pipeline-specific annotations signify that the while loop will be transformed into a parallel pipeline. Since *DetectFaces* is the most computationally intensive stage, we replicate it by the number of estimated free processing units. The desired goal is to increase the number of processed images per time unit, which translates to the maximization of pipeline throughput.

6.2.3 Transformed Code

This high-level representation is transformed by the transformation tool and extended into a representation that utilizes the coordination layer for pipelining on top of the StarPU heterogeneous runtime system to coordinate and schedule the execution on concrete target processing units. All component implementation variants are queried from the PEPPHER component repository. Besides CPU implementation variants, *ConvertAndResize* and *DetectFaces* components feature an additional GPU implementation variants with CUDA support, thus allowing the utilization of available NVidia GPUs . Listing 6.3 shows the relevant parts of the transfomed code.

```
/* internal requirements */
#include <vpattern.h>

/* components - application specific */
#include "Common/common.hh"

using namespace vtt;
...
/* Stage wrapper funciton(s) */
void StageWrapper1 (void *descr[], void *args) {
    char *__vtt_temp0 = (char *) STARPU_VECTOR_GET_PTR(descr[0]);
    unsigned int __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[0]);
    uchar *__vtt_temp1 = (uchar *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[1]);

    ReadImage(__vtt_temp0, __vtt_temp0_size, __vtt_temp1, __vtt_temp1_size);
}

...
```

```
/* while loop wrapper funciton(s) */
bool WhileLoopWrapper() {
  if ( inputstream >> file ) {
    return true;
  } else {
    /* set termination trigger */
    return false;
  }
}
...
int main(int argc, char *argv[]) {
  ...
  /* create a new pipeline  */
  Pipeline __vtt_pipeline;

  /* set up a new pipeline manager */
  PipelineManager __vtt_manager;

  /* set up a new runtime configuration */
  RuntimeConfiguration __vtt_rconf(STARPU_RUNTIME, NCPUS, NGPUS, SCHEDULER);

  /* set up pipeline stages */
  Stage *s1 = pipeline.CreateStage("facedetect_stage1", STAGE_TYPE_SOURCE);
  Stage *s2 = pipeline.CreateStage("facedetect_stage2", STAGE_TYPE_MIDDLE);
  Stage *s3 = pipeline.CreateStage("facedetect_stage3", STAGE_TYPE_MIDDLE);
  Stage *s4 = pipeline.CreateStage("facedetect_stage4", STAGE_TYPE_SINK);

  s1->AddParameter((void *)&file, VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);
  s1->AddParameter(VTT_UCHAR, img_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s2->AddParameter(VTT_UCHAR, img_size, STAGE_PARAMETER_ACCESS_MODE_READ);
  s2->AddParameter(VTT_UCHAR, out_img_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s3->AddParameter(VTT_UCHAR, out_img_size, STAGE_PARAMETER_ACCESS_MODE_READWRITE);

  s4->AddParameter(VTT_UCHAR, out_img_size, STAGE_PARAMETER_ACCESS_MODE_READ);
  s4->AddParameter((void *)&file, VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);

  /* connect the stages */
  pipeline.ConnectStages(s1, s2);
  pipeline.ConnectStages(s2, s3);
  pipeline.ConnectStages(s3, s4);

  s1->add_functionality(StageWrapper1, STAGE_IMPLEMENTATION_CPU);
  s1->SetControlFunction(WhileLoopWrapper);
  s2->add_functionality(StageWrapper2, STAGE_IMPLEMENTATION_CPU);
  s3->add_functionality(StageWrapper3, STAGE_IMPLEMENTATION_CPU);
```

```
#ifdef vtt_CUDA_CAPABLE
  s3->add_functionality(StageWrapper3_GPU, STAGE_IMPLEMENTATION_CUDA);
#endif
  s4->add_functionality(StageWrapper4, STAGE_IMPLEMENTATION_CPU);

  /* Adjust replication factors */
  s3->set_replication_factor(rf);

  /* associate pipeline with the pipeline manager and the StarPU runtime system */
  manager.init(&pipeline, conf2, false););

  /* run the pipeline */
  manager.RunPipeline(&pipeline);
  ...
  /* query pipeline performance data */
  manager.pipeline(0)->ExecutionTime();
  ...
}
```

Listing 6.3: Face Detection: the transformed code parts.

6.2.4 Results

This section presents performance measurements of face detection experiment carried out on NADIA and PHIA systems. The test application processes a series of 500 images in HD (1280x720 pixels) resolution (also referred to as "720p"). Each image in this data set contains an arbitrary number of human faces. As the face detection algorithm used in the detection stage relies on pre-trained classifiers, we used the cascade classifiers for human faces that ship with the OpenCV library. We measured the performance of the code that utilizes the coordination layer for pipelining ("PEPPHER Pipeline"), as well as the baseline versions of the same algorithm ("OpenCV Baseline"). The baseline versions utilize the same high-level while loop with similar PEPPHER components, however parallel execution and utilization of GPUs relies on the facilities provided by the OpenCV library.

Table 6.1 presents the results achieved with the baseline Face Detection implementation using OpenCV compiled with and without GPU or Intel TBB support.

Resource Utilization	Execution Time (sec)	
	NADIA	PHIA
1 CPU Core + 1 GPU	75.384	79.269
Whole CPU (OpenCV with TBB)	124.344	113.730
1 CPU Core	505.849	427.231

Table 6.1: Face Detection: Comparison of different baseline versions on NADIA and PHIA systems.

Figure 6.1 shows PHIA system speedup results relative to the fastest OpenCV baseline version. The blue bars represent the results of different baseline versions. The first blue bar on the left shows the slowest performing baseline version using only 1 CPU core. The second one denotes the version that used all 16 available CPU cores on the PHIA system. The fastest executing baseline version, which utilized 1 CPU core and 1 GPU, is represented by the third blue bar.

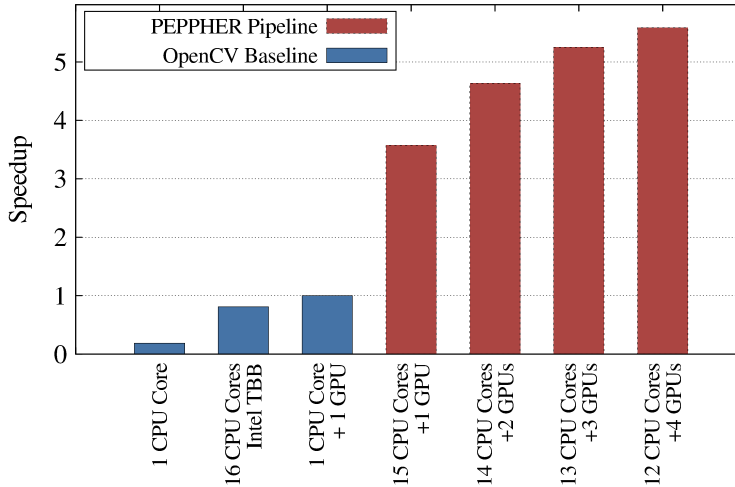


Figure 6.1: Face Detection: Speedup results relative to the fastest performing baseline version on the PHIA system.

The red bars represent the results of the PEPPHER parallel pipeline version executed with different configuration parameters (stage replication factors, number of CPU cores, number of GPUs , stage buffer sizes and runtime scheduling strategies). We present configurations that achieve hybrid execution using all available CPU cores combined with varying number of GPUs (from one up to four). The first red bar shows a speedup of 3.5 relative to the fastest baseline version. This configuration used only 1 GPU in combination with 15 CPU cores. We see less drastic changes as more GPU units are used simultaneously, resulting in speedups of 4.6, 5.3 and 5.9 for configurations that used two, three and four GPUs respectively.

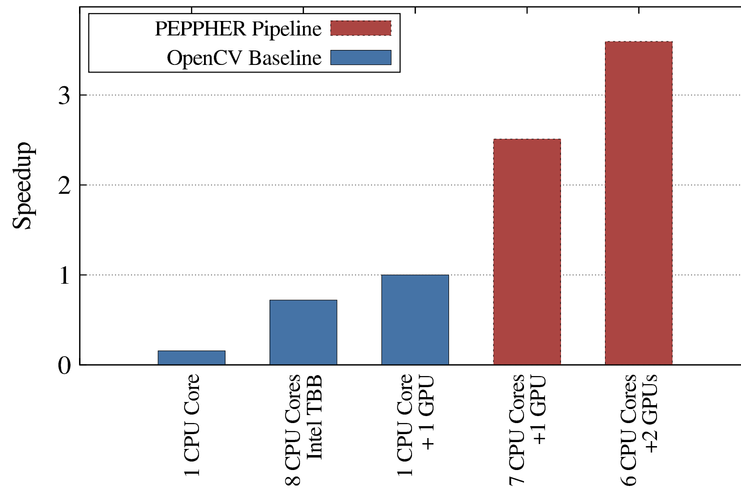


Figure 6.2: Face Detection: Speedup results relative to the fastest performing baseline version on the NADIA system.

Speedup results for the NADIA system are shown in Figure 6.2. Similarly, the third blue bar represents the fastest OpenCV baseline version. The best performing PEPPHER Pipeline version on NADIA used 6 CPU cores and 2 GPUs and resulted in an overall speedup of 3.6, whilst using 7 CPU cores with 1 GPU resulted in a speedup of 2.5.

The influence of stage replication factors on performance can be observed in Figure 6.3. The results indicate that increasing the replication factor of the longest running stage may have a significant impact on performance as it improves the overall balance of the pipeline. This figure shows impact of the replication factor of the slowest stage (*DetectFaces*) on the PHIA system.

The execution times on this figure have been normalized to the longest execution time. The blue bars represent PEPPHER Pipeline that tries to utilize up to four GPUs with the replication factor set to 1. As it can be observed, low replication factors may limit potentially exploitable parallelism and prevent efficient usage of more execution units at the time. The figure shows roughly the same execution times regardless of the number of GPUs in use.

On the contrary, carefully adjusted replication factors may enable utilization of other available execution units in the system. The red bars describe this behavior. The first red bar shows relative execution with 15 CPU cores and 1 GPU. Even though the same configuration is also used with replication factor 1, we observe significantly faster execution due to the fact that the rest of CPU cores are also being utilized. In the case when replication factor is set to 16, we record additional performance gains when more GPUs are available.

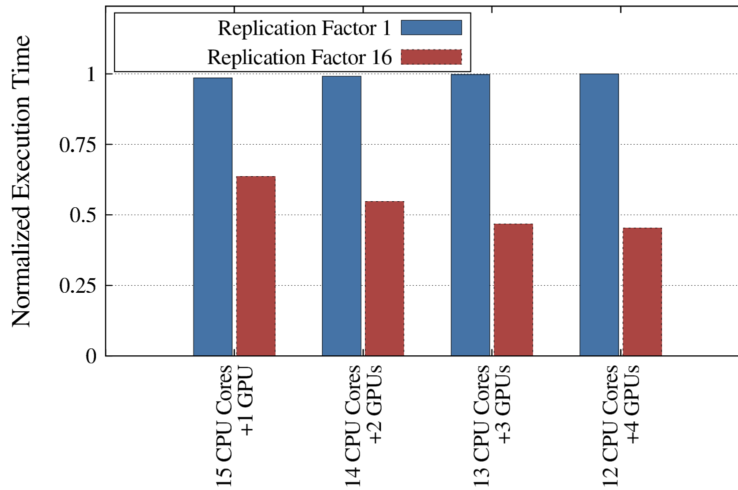


Figure 6.3: Face Detection: The impact of replication factors on the overall performance. The blue bars show the effect of low replication factors that may limit potentially exploitable parallelism. Carefully adjusted replication factors will enable the framework to utilize more computational units, as shown by the red bars.

6.2.5 Additional Experiments

In this section we compare a hand-written version of the program that directly utilizes the StarPU runtime system to a version using the coordination layer for pipelining. The experiments have been conducted on the CORA system. The data set in use is slightly different featuring 3425 images in 640x480 resolution. Figure 6.4 shows comparison of two different implementations of the face detection application. The blue bars represent optimized hand-coded versions that directly utilize the StarPU runtime system, while the red bars represent versions that use the pipeline coordination layer.

The performance results indicate that the overhead caused by the pipeline coordination layer is insignificant. Sometimes the generated code even outperforms the hand-written code. It should be noted that utilizing the StarPU runtime system directly is much more complex than using the PEPPHER pipeline annotations. The figure shows the same effect, but considers different scheduling policies (HEFT and EAGER), which are supported by the StarPU runtime. We observe that the heterogeneous earliest finish time (HEFT) policy relying on well trained performance models, performs significantly better compared to EAGER that uses a simple greedy algorithm to schedule the tasks.

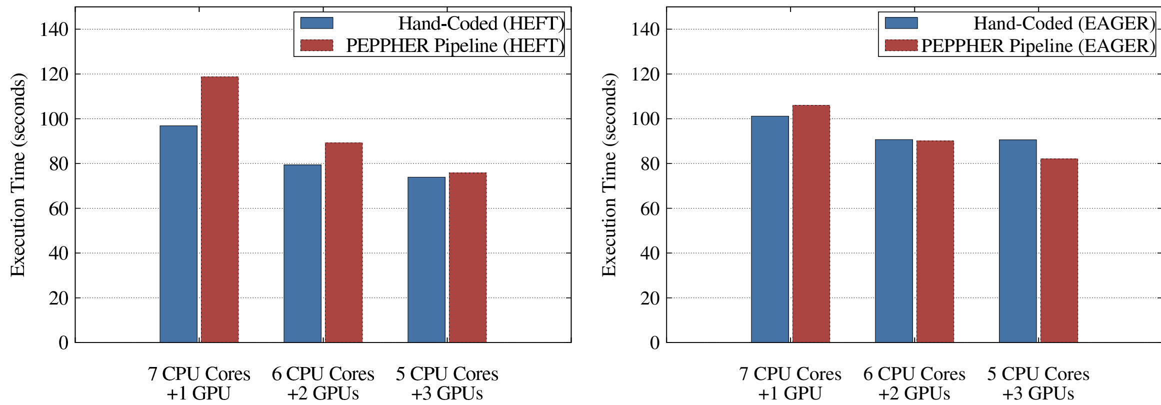


Figure 6.4: Face Detection: Hand-coded version vs PEPPHER Pipeline [48]. The blue bars represent optimized hand-coded version that utilize the StarPU runtime system directly. The red bars correspond to a version that uses the pipeline coordination layer. Figures (a) shows the effect when HEFT scheduling policy is in use, while Figure (b) shows the effect when EAGER scheduling policy is used.

6.3 Needleman-Wunsch Application

Rodinia Benchmark Suite [49] provides a set of benchmarks with an aim to provide an evaluation means for heterogeneous systems. The suite ships with the number of applications from various domains of computer science, such as bioinformatics, data mining, and image processing [50]. Often, the benchmarks feature implementation variants written in OpenMP, OpenCL and/or CUDA. For the evaluation purposes, we have selected an application that comes from the domain of bioinformatics and uses Needleman-Wunsch algorithm to perform non-linear global optimization for DNA, RNA or protein sequence alignments.

The original code from the Rodinia suite was adapted to process a stream of sequence pairs, utilizing implementation variants available for the alignment. The resulting code features four-stage pipeline, consisting of stages for reading the sequence from a file (*ReadSequence(...)*), initialization of data structures in use (*PrepareSequence*), the main sequence alignment algorithm (*Process*) with sequential, OpenCL and CUDA implementation variants, and finally a stage for performing the traceback and writing the results to an output file (*Traceback*). A high-level Needleman-Wunsch code is shown in Listing 6.4.

```
...
int main(int argc, char *argv[]) {
...
    while ( *inputstream >> filename ) {
        ReadSequence( file, 256, input_itemsets, full_size );
        PrepareSequence( input_itemsets, full_size , reference, full_size  );
        Process( input_itemsets, full_size, reference, full_size );
        Traceback( file, 256, input_itemsets, full_size, reference, full_size);
    }
...
}
```

Listing 6.4: Needleman-Wunsch: Application utilizes the PEPPHER components (re-engineered from Rodinia Benchmark Suite) to perform sequence alignment.

6.3.1 Annotated Code

The high-level code annotated with the PEPPHER pipeline directives is shown in Listing 6.5.

```
...
    #pragma pph pipeline buffer(UNORDERED, 32)
    while ( *inputstream >> filename ) {
        ReadSequence( file, 256, input_itemsets, full_size );
        PrepareSequence( input_itemsets, full_size , reference, full_size  );
        #pragma pph stage replication(rf)
        Process( input_itemsets, full_size, reference, full_size );
        Traceback( file, 256, input_itemsets, full_size, reference, full_size);
    }
...
```

Listing 6.5: Needleman-Wunsch: The high-level pipeline directives signifying the PEPPHER pipeline and setting the replication factors for compute intensive stage.

The *pph pipeline* directive signifies that the while loop will be transformed into a PEPPHER pipeline featuring bounded buffers with the capacity of 32 elements, and utilizing *unordered* queuing mechanisms. Additionally, the most computationally intensive component *Process* should be replicated by a predefined replication factor `rf`.

6.3.2 Transformed Code

The high-level representation shown in Listing 6.5 is transformed into a representation that utilizes the coordination layer for pipelining. All component implementation variants are queried from the PEPPHER component repository. The *Process* component features additional CUDA and OpenCL implementation variants, thus allowing the utilization of available CUDA/OpenCL enabled GPUs and/or other OpenCL-enabled accelerators supported by the runtime system. Listing 6.6 shows the relevant parts of the transformed code generated by the coordination layer.

```
#include <vpattern.h>
...
/* Stage wrapper funciton(s) */
void StageWrapper1 (void *descr[], void *args) {
    char *__vtt_temp0 = (char *) STARPU_VECTOR_GET_PTR(descr[0]);
    unsigned int __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[0]);
    int *__vtt_temp1 = (int *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[1]);

    ReadSequence(__vtt_temp0, __vtt_temp0_size,
                 __vtt_temp1, __vtt_temp1_size);
}
void StageWrapper2 (void *descr[], void *args) {
    int *__vtt_temp0 = (int *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[1]);
    int *__vtt_temp1 = (int *) STARPU_VECTOR_GET_PTR(descr[2]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[2]);

    PrepareSequence(__vtt_temp0, __vtt_temp0_size,
                    __vtt_temp1, __vtt_temp1_size);
}
void StageWrapper3 (void *descr[], void *args) {
    int *__vtt_temp0 = (int *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[1]);
    int *__vtt_temp1 = (int *) STARPU_VECTOR_GET_PTR(descr[2]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[2]);

    Process(__vtt_temp0, __vtt_temp0_size,
            __vtt_temp1, __vtt_temp1_size);
}
#ifdef vtt_OPENCL_CAPABLE
void StageWrapper3_opencl (void *descr[], void *args) {
    cl_mem __vtt_temp0 = (cl_mem) STARPU_VECTOR_GET_DEV_HANDLE(descr[1]);
    uint32_t __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[1]);
    cl_mem __vtt_temp1 = (cl_mem) STARPU_VECTOR_GET_DEV_HANDLE(descr[2]);
```

```
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[2]);

    Process_opencl(__vtt_temp0, __vtt_temp0_size,
                   __vtt_temp1, __vtt_temp1_size);
}
#endif
#ifdef vtt_CUDA_CAPABLE
void StageWrapper3_cuda (void *descr[], void *args) {
    int *__vtt_temp0 = (int *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp0_size = STARPU_VECTOR_GET_NX(descr[1]);
    int *__vtt_temp1 = (int *) STARPU_VECTOR_GET_PTR(descr[2]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[2]);

    Process_cuda(__vtt_temp0, __vtt_temp0_size,
                 __vtt_temp1, __vtt_temp1_size);
}
#endif
void StageWrapper4 (void *descr[], void *args) {
    char *__vtt_temp0 = (char *) STARPU_VECTOR_GET_PTR(descr[0]);
    int __vtt_temp0_size = (int)STARPU_VECTOR_GET_NX(descr[0]);
    int *__vtt_temp1 = (int *) STARPU_VECTOR_GET_PTR(descr[1]);
    uint32_t __vtt_temp1_size = STARPU_VECTOR_GET_NX(descr[1]);
    int *__vtt_temp2 = (int *) STARPU_VECTOR_GET_PTR(descr[2]);
    uint32_t __vtt_temp2_size = STARPU_VECTOR_GET_NX(descr[2]);

    Traceback(__vtt_temp0, __vtt_temp0_size,
              __vtt_temp1, __vtt_temp1_size,
              __vtt_temp2, __vtt_temp2_size);
}
...
int main(int argc, char *argv[]) {
  ...
  /* create a new pipeline with with unordered buffers between the stages */
  Pipeline __vtt_pipeline(BUFFER_ORDER_UNORDERED);

  /* set up a new pipeline manager */
  PipelineManager __vtt_manager;

  /* set up a new runtime configuration */
  RuntimeConfiguration __vtt_rconf(STARPU_RUNTIME, NCPUS,
                                   NGPUS, SCHEDULER);

  /* set up pipeline stages */
  if (verbose) cout << "vtt: Creating stages ..." << endl;
  Stage *s1 = pipeline.CreateStage("facedetect_stage1", STAGE_TYPE_SOURCE);
  Stage *s2 = pipeline.CreateStage("facedetect_stage2", STAGE_TYPE_MIDDLE);
  Stage *s3 = pipeline.CreateStage("facedetect_stage3", STAGE_TYPE_MIDDLE);
  Stage *s4 = pipeline.CreateStage("facedetect_stage4", STAGE_TYPE_SINK);
```

```
  s1->AddParameter((void *)&file, VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);
  s1->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s2->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_READWRITE);
  s2->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_WRITE);

  s3->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_READWRITE);
  s3->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_READWRITE);


  s4->AddParameter(VTT_CHAR, 256, STAGE_PARAMETER_ACCESS_MODE_READ);
  s4->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_READ, 1);
  s4->AddParameter(VTT_INT, full_size, STAGE_PARAMETER_ACCESS_MODE_READ, 1);

  /* connect the stages */
  pipeline.ConnectStages(s1, s2);
  pipeline.ConnectStages(s2, s3);
  pipeline.ConnectStages(s3, s4);

  s1->add_functionality(StageWrapper1, STAGE_IMPLEMENTATION_CPU);
  s1->SetControlFunction(WhileLoopWrapper);
  s2->add_functionality(StageWrapper2, STAGE_IMPLEMENTATION_CPU);
  s3->add_functionality(StageWrapper3, STAGE_IMPLEMENTATION_CPU);
#ifdef vtt_CUDA_CAPABLE
  s3->add_functionality(StageWrapper3_GPU, STAGE_IMPLEMENTATION_CUDA);
#endif
#ifdef vtt_OPENCL_CAPABLE
  s3->add_functionality(StageWrapper3_GPU, STAGE_IMPLEMENTATION_OPENCL);
#endif
  s4->add_functionality(StageWrapper4, STAGE_IMPLEMENTATION_CPU);

  /* Adjust replication factors */
  s3->set_replication_factor(rf);

  /* associate pipeline with the pipeline manager and the StarPU runtime system */
  manager.init(&pipeline, conf2, false););

  /* run the pipeline */
  manager.RunPipeline(&pipeline);



  /* query pipeline performance data */
  manager.pipeline(0)->ExecutionTime();
  ...
}
```

Listing 6.6: Needleman-Wunsch: the transformed code parts.

6.3.3 Results

In this section we present performance measurements of Needleman-Wunsch experiment conducted on PHIA and NADIA systems. For testing purposes we have generated a data set of 500 pairs of sequences, featuring lengths of up to 4096 elements. Table 6.2 presents the results achieved with the baseline Needleman-Wunsch implementation using components re-engineered from the Rodina benchmark.

Resource Utilization	Execution Time (sec)	
	NADIA	PHIA
1 CPU Core + 1 GPU	50.441	69.182
Whole CPU (OpenMP)	163.846	205.979
1 CPU Core	342.810	355.987

Table 6.2: Needleman-Wunsch: Comparison of different baseline versions on NADIA and PHIA systems.

Figure 6.5 shows speedup results on the PHIA system. The blue bars represent the baseline versions of the algorithm. The fastest baseline execution was achieved with 1 CPU core and 1 GPU, resulting with a speedup of 3, relative to the baseline version that uses all 16 CPUs. In comparison to the Face Detection results (Figure 6.1), the GPU performance gains over CPU execution are slightly more pronounced. In fact, the actual algorithm executed roughly 5 times faster on a single GPU then on a single CPU core on the PHIA system.

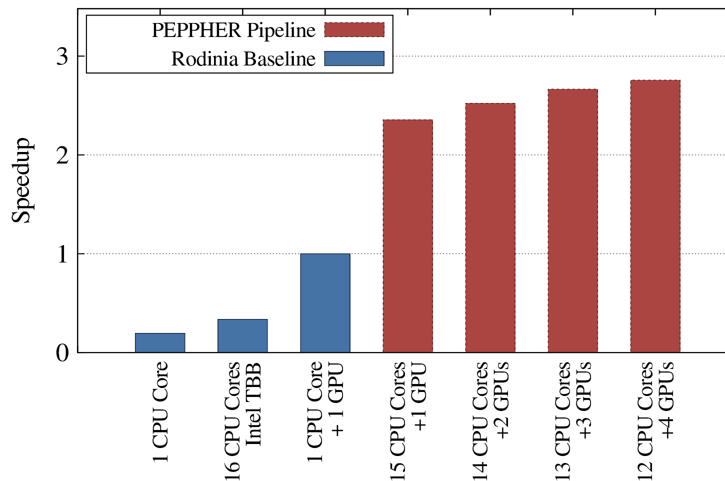


Figure 6.5: Needleman-Wunsch: Speedup results relative to the fastest performing baseline version on the PHIA system.

The first red bar on the figure shows a speedup of 2.4 relative to the fastest baseline version. The maximum achieved speedup of 2.8, represented by the red bar on the right side, used 12 CPU cores and 4 GPUs . Interestingly, the parallel pipeline version of Needleman-Wunsch algorithm yields a slightly lower performance gains compared to the Face Detection example. This may be attributed to the particular data set in use, since it directly affects the ratio of data transfer time to computation time.

On NADIA, only minor performance gains were observed. A speedup of 1.5 was measured when 2 GPUs were used together with 6 CPU cores. This is shown in Figure 6.6.

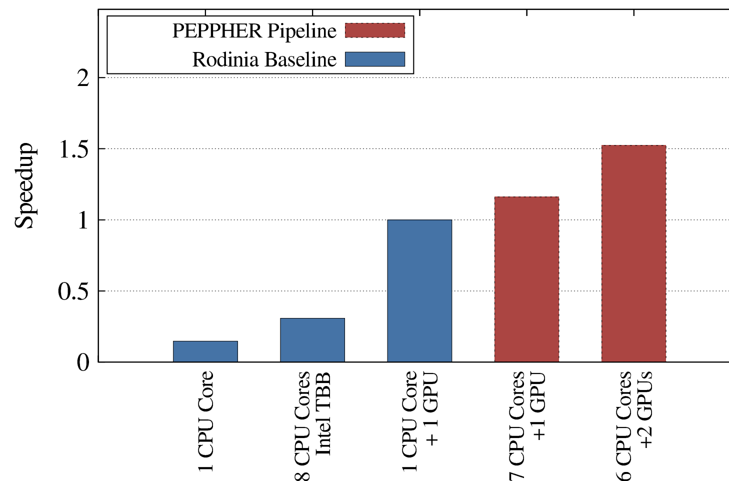


Figure 6.6: Needleman-Wunsch: Speedup results relative to the fastest performing baseline version on the NADIA system.

6.4 Summary

We have evaluated our approach using two applications from different domains of computer science. The high-level coordination language combined with component-based, task-parallel approach enabled execution of parallel pipeline without having to deal with low level issues. The initial performance results of the current prototype are encouraging and confirm viability of our approach and yet leave space for further improvements. We have discovered that even though default values and general rules tend to use all available execution units in a system, finding the best values for different pipeline-specific parameters (stage replication factors, input/output buffer types and sizes for each stage in the pipeline) or runtime-specific

parameters (task scheduling policy, number of CPUs/GPUs in use) and optimizing performance can be challenging for larger pipelines. On the one hand, determining the slowest stage in the pipeline, may not always be obvious and simply using large replication factors can lead to thread over-subscription and performance degradation. On the other hand, modest performance improvements for larger number of GPUs might drive the decision to avoid usage of the whole system and optimize the execution for power consumption. Such issues are addressed in somewhat greater details in Section 8.

During the development we have discovered that extracting parallel programming patterns from larger libraries can be challenging. Often, libraries are abstracting internal functionalities in order to provide a programmer with simpler interfaces. While this way of hiding complexity is usually beneficial, exposition of relevant implementation parts could reduce amount of re-engineering effort required to fully utilize the other libraries that operate on lower level.

Although the transformed code can benefit from utilizing heterogeneous architectures, the evaluated applications consist of rather simple four-stage pipelines, hence reducing potential optimization opportunities. In the future evaluation work we would like to investigate applicability of this approach on more complex algorithms and use cases (such as face recognition). These algorithms could be represented by more complex pipelines, thus providing opportunities for the coordination layer for pipelining and underlying runtime system to apply further optimizations. Additionally, we plan to extend the coordination layer to support a wider range of parallel programming patterns, which would enable similar benefits for a broader variety of applications.

7 Related Work

The potential computational power and better power utilization in hybrid systems are driving the evolution of the software that aims to ease the complexity programmers need to deal with. A significant effort has been invested on many levels of the software stack to address the heterogeneity - from compilers, low-level parallelism (such as multithreading), libraries (e.g., BLAS [51], MAGMA [52], PLASMA [53], etc.), runtime systems, to parallel programming models/ languages (such as Cilk [21], OpenCL [54], etc.), and autotuners. However, there is still no single solution to this problem and programmers are often forced to utilize different programming models, libraries and compilers to increase overall utilization of the system. The work on the parallel patterns and skeletons [55, 56, 57, 2, 58] that is used in combination with these approaches are of significant relevance to this thesis.

The multithreading approaches (e.g: POSIX threads) allow fine grained control over thread execution and synchronization. However, they have proven to be very difficult to work with, especially for large scale applications. OpenMP [10] addresses multithreading at higher level and uses compiler directives in the combination with well-known programming languages (C/C++/Fortran) to express parallelism, thus allowing reuse of existing code. Traditionally, OpenMP targeted homogeneous architectures and supported a shared memory model. Recent extensions for heterogeneous architectures (also know as the OpenMP accelerator model) in the OpenMP 4.0 specification [59] aim to support accelerator units as well. However, the work on the support for pipeline patterns [58, 60] does not address heterogeneous execution, and the support for heterogeneous architectures in the new OpenMP standard is still in its early phase [61].

Another related approach is OmpSs [62] that builds on the StarSs [63] programming model and combines it with the ideas from OpenMP to support asynchronous execution and heterogeneity. However, there is no support for pipelining or other parallel patterns. Intel Thread Building Blocks (TBB) is a C++ template library that builds on top of a task-based programming model [46]. It utilizes work-stealing tasks schedulers to map tasks to under-

lying architecture and perform load balancing among employed execution units. Although there has been some work towards the support for heterogeneous systems equipped with Xeon Phi's [64, 65, 66], the Intel TBB library does not yet fully support hybrid architectures (e.g., no GPU support). Moreover, the pipeline patterns used in TBB, do not utilize multiple kernel implementation variants for pipeline stages.

The PGI Accelerator compilers [67] and HMPP [22] build on the OpenACC standard [68]. This standard describes a set of directives that can be used to annotate code regions to be offloaded to an accelerator. Such set of directives can be used to extend standard and widely adopted programming languages, such as C, C++ and Fortran. However, neither of these approaches provides support for pipeline patterns. The feedback-directed pipeline (FDP) parallelization framework [69] does the stage-to-core mapping according to measured throughput of the stages, thus dynamically assigning more resources to more computationally intensive stages and addressing power efficiency by re-targeting non-limiter stages without performance degradation. However, no multiple implementation variants for stages are supported.

StreamIt is a domain specific programming language and a compiler designed for streaming applications [24]. The central concept in StreamIt are *filters* which represent basic units of computation and may be composed into a communication network [24]. The StreamIt supports pipeline specific optimisations, however the support for heterogeneous architectures is not addressed.

The work from Schaefer at al. [70, 71] presents an architecture description language that supports expression of parallel programming patterns and combines it with auto-tuning. Although the support for pipeline patterns is present, the support for heterogeneous architectures is not addressed [34].

8 Conclusion and Future Work

The shift towards many-core hardware architectures is driven by the capabilities and limitations of semiconductor manufacturing process [72]. However, developing software that can utilize potential benefits of such hardware proves to be considerably complex. In addition to parallelism, programmers have to cope with an increasing heterogeneity of underlying systems that may comprise many cores of different type, or accelerators such as GPUs or Xeon Phis, etc. Hence, many new programming models/libraries (such as Intel TBB, StreamIt, STAPL, Cilk++, etc.) aim to make parallel programming easier and less error prone by employing higher-level abstractions for parallelism and providing facilities for concurrent execution. Nevertheless, efficient programming of new systems still remains difficult, parallel patterns may be hard to express and optimal performance is often all but reached.

This thesis presented the work on the framework for supporting high-level pipeline patterns on heterogeneous many-core architectures. It has been developed within the context of three-year European project PEPPHER and it relies on a component model and execution model developed within the same project. The framework consists of a high-level coordination language, a source-to-source compiler and the coordination layer for pipelining that runs on top of a dynamic heterogeneous runtime system. The high-level coordination language adds the support for expression of parallel pipeline patterns in the form of annotations (C/C++ directives) for sequential C/C++ codes that use the PEPPHER multi-architectural components (see Section 3.3). Based on this annotated code, the source-to-source compiler is able to transform a user-annotated high-level code into the representation that utilizes the coordination layer for pipelining. Essentially, the layer is an object-oriented C++ library, where C++ classes are used to represent relevant pipeline structural properties and functionalities, as well as to orchestrate parallel execution when possible. It relies on a component-based approach, where pipeline stages correspond to multi-architectural components that comprise implementations tailored for different types of execution units. Depending on the pipeline structure and user-provided information, the coordination layer interacts with the heterogeneous runtime system (StarPU) by dynamically creating and submitting the tasks

to it. Based on the information such as the selected scheduling policy, available component implementation variants, data transfer costs, performance models based on historical execution profiles and current system load, StarPU executes the tasks on the available execution units of the underlying hardware with an aim to minimize overall execution.

The usage of the high-level parallel patterns in a combination with such highly dynamic heterogeneous runtime system is of a particular interest to this work. On the one hand, using high-level patterns rises the level of abstraction that programmers have to deal with, thus making code easier to write. The coordination layer is able to provide a set of hints to the runtime system for a particular parallel pattern and expose parameters that can be used for further tuning of execution and controlling of memory footprint. On the other hand, runtimes such as StarPU, already support a lot of performance optimizations internally (optimizations of data transfer, task data dependencies, execution, performance models, historic performance data, etc.), making it unclear whether the interference with such systems can bring significant benefits in the general case.

The current implementation of the framework supports parallel pipeline patterns. The results shown in Chapter 6 are encouraging. We have used two applications from the area of the computer vision and bioinformatics and executed them on two different hybrid many-core systems. The achieved performance is comparable to manual parallelization, although significantly higher-level of abstraction is used.

In the future, we aim to introduce the support for other high-level parallel patterns such as map-reduce or master-worker pattern. Additionally, we would like to investigate the applicability of this approach on more complex algorithms and use cases (e.g. face recognition), where more complex pipeline could provide opportunities for the PEPPHER pipeline coordination layer and the PEPPHER runtime system to perform further optimizations.

Presently, the framework is being extended in the direction of automatic online tuning. In the context of three-year European project AutoTune [73, 74, 75, 76, 77, 78], we are developing autotuning techniques for optimizing high-level pipeline patterns for heterogeneous systems. For this purpose, the framework is being extended to expose various tuning parameters for the external tuners, in particular Periscope Tuning Framework (PTF) [73, 35]. We distinguish between three logical groups of parameters that target different parameters of execution. The first group addresses pattern-specific characteristics such as stage replication factors for each stage in the pipeline, and input and output buffer size. The second group

aims to describe machine-specific parameters, such as the number of CPU cores, hardware threads or the number of GPUs. The third group targets runtime-specific parameters such as the scheduling policy that should be utilized by the runtime system. Additionally, we want to utilize the performance data collected by the coordination layer to automatically detect performance-limiting stages in the pipeline and adjust pipeline-specific tuning parameters to improve overall execution.

```
#pragma pph pipeline with buffer (UNORDERED , ?)
while ( inputstream >> file ) {
  func1 ( file , data );
  #pragma pph stage replicate (2:10:2)
  func2 ( data , cdata );
  func3 ( file, cdata );
}
```

Listing 8.1: User-provided hints for external tuners. When constructing a search space, the plugin will take into a consideration user-provided ranges (*min*, *max*, *step*) if they are available. This example shows how replication factors can be hinted [77].

This approach is backed up with extensions to the high-level coordination language that allows programmers to provide hints to the tuner, and generally enhance user interaction with the tuner. Such user-provided hints may contain the user-defined tuning ranges that aim to reduce potentially huge search space and speed-up the tuning process. For example, in Listing 8.1 a range of [2:10:2] is specified for the replication factor in the form of *min*, *max* and *step*, and the “?” symbol indicates that the buffer size should be automatically adjusted by the tuner.

List of Figures

1.1 Moore's Law . . . 3
1.2 Top500 List: Peak and sustained performance since 1993 . . . 4
1.3 High-level representation of homogeneous multi-core . . . 5
1.4 High-level representation of heterogeneous multi-core . . . 6

2.1 PEPPHER Software Stack . . . 15
2.2 PEPPHER Components . . . 16
2.3 PEPPHER Execution Model . . . 21

3.1 Pipeline Pattern: Illustration of non-pipelined execution . . . 28
3.2 Pipeline Pattern: Illustration of pipelined execution . . . 29
3.3 Pipeline Pattern: Imbalanced pipeline illustration . . . 29
3.4 Pipeline Pattern: Linear pipeline illustration . . . 30
3.5 Pipeline Pattern: Non-linear pipeline illustration . . . 31

4.1 Coordination Layer in the PEPPHER framework . . . 37
4.2 Coordination Layer: Buffer creation policies . . . 43
4.3 Coordination Layer: Stage execution mechanism . . . 47
4.4 Coordination Layer: Buffer class diagram . . . 48
4.5 Coordination Layer: Execution model sequence diagram . . . 56
4.6 Coordination Layer: Class diagram of the *Runtime* class . . . 59
4.7 Coordination Layer: Performance measurements for replicated stages. . . . 61
4.8 Coordination Layer: Performance measurements for non-replicated stages . 62

5.1 Transformation process: High-level overview . . . 66

6.1 Face Detection: Speedup results on PHIA . . . 82
6.2 Face Detection: Speedup results on NADIA . . . 83
6.3 Face Detection: Impact of replication factors . . . 84
6.4 Face Detection: Effect of scheduling policies . . . 85

6.5 Needleman-Wunsch: Speedup results on PHIA 90
6.6 Needleman-Wunsch: Speedup results on NADIA 91

List of Tables

1.1 Software and hardware technologies used in the thesis 9
1.2 Software developed in the thesis 10

6.1 Face Detection: Comparison of different baseline versions 82
6.2 Needleman-Wunsch: Comparison of different baseline versions 90

Abbreviations

APU Accelerated Processing Unit. 6

AST abstract syntax tree. 66, 67, 69

BLAS Basic Linear Algebra Subprograms. 93

CBSE Component Based System Engineering. 14

Cell BE Cell Broadband Engine. 3, 4, 6, 8

CPU Central Processing Unit. 2–6, 9, 15, 19, 28, 46, 58, 62, 72, 75–77, 79, 82, 83, 90–92, 97

CUDA Compute Unified Device Architecture. 8, 19, 20, 40, 46, 58, 76, 85, 87

DAG directed acyclic graph. 7, 20, 57, 68, 71

DMA Direct Memory Access. 5

DMDA deque model data aware. 58

DNA deoxyribonucleic acid. 40, 85

GPGPU General-purpose Computing on Graphics Processing Units. 8

GPU Graphics Processing Unit. i, iii, 3, 4, 6, 9, 15, 19, 20, 28, 58, 62, 72, 75–77, 79, 81–83, 87, 90–92, 94, 95, 97

HD high definition. 81

HEFT heterogeneous earliest finish time. 58, 84, 85

HMPP Hybrid Multicore Parallel Programming. 7, 94

HPC High Performance Computing. 4

HPF High Performance Fortran. 7

ILP instruction-level parallelism. 2, 5

IR intermediate representation. 65, 66

MPI Message Passing Interface. 7

OpenCL Open Computing Language. 8, 20, 40, 46, 58, 85, 87, 93

OpenCV Open Source Computer Vision. 9, 15, 19, 20, 77, 78, 81–83

OpenMP Open Multi-Processing. 5, 9, 40, 85, 90, 93

PDL Platform Description Language. 18

PLASMA Parallel Linear Algebra for Scalable Multi-core Architectures. 93

POSIX Portable Operating System Interface. 93

RNA ribonucleic definition. 85

TBB Thread Building Blocks. 9, 77, 81, 82, 93–95

UPC Unified Parallel C. 7, 9, 66

XML Extensible Markup Language. 15–17

Bibliography

[1] Herb Sutter, „A Fundamental Turn Toward Concurrency in Software“, *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16–23, 2005, ISSN: 1044-789X (cit. on p. 1).

[2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*, 1st. Addison-Wesley Professional, 2004, ISBN: 0321228111 (cit. on pp. 1, 5, 7, 29, 31, 93).

[3] Gordon E. Moore, „Cramming More Components Onto Integrated Circuits“, *Electronics*, vol. 38, no. 8, 1965 (cit. on p. 2).

[4] Manek Dubash. (2006). Moore's Law is dead, says Gordon Moore, [Online]. Available: http://www.techworld.com/opsys/news/index.cfm?NewsID=3477%22 (cit. on p. 2).

[5] Gordon Moore. (2012). Moore's Law web page at Intel, [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html (visited on 06/07/2012) (cit. on p. 2).

[6] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, Eds., *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ISBN: 1-55860-871-0 (cit. on p. 2).

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, „The Landscape of Parallel Computing Research: A View from Berkeley“, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006 (cit. on p. 2).

[8] Top500.org. (2013). Top500 Supercomputing Sites - Website, [Online]. Available: http://www.top500.org (visited on 06/25/2014) (cit. on p. 3).

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006, ISBN: 0123704901 (cit. on p. 5).

[10] OpenMP Architecture Review Board, „OpenMP Application Program Interface“, Specification, 2011 (cit. on pp. 5, 93).

[11] J. A. Kahle, M. N. Day, P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, „Introduction to the Cell multiprocessor“, *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 589–604, 2005, ISSN: 0018-8646 (cit. on p. 6).

[12] A. Branover, D. Foley, and M. Steinman, „AMD Fusion APU: Llano“, *Micro, IEEE*, vol. 32, no. 2, pp. 28–37, 2012, ISSN: 0272-1732 (cit. on p. 6).

[13] *Intel® Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, 2012 (cit. on p. 6).

[14] Andrs Vajda, *Programming Many-Core Chips*, 1st. Springer Publishing Company, Incorporated, 2011, ISBN: 1441997385, 9781441997388 (cit. on pp. 6, 7).

[15] High Performance Fortran Forum (HPFF), *High Performance Fortran Language Specification*, 1997 (cit. on p. 7).

[16] UPC Consortium, „UPC Language Specifications, v1.2“, Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005 (cit. on p. 7).

[17] R. W. Numrich and J. Reid, „Co-array Fortran for Parallel Programming“, *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998, ISSN: 1061-7264 (cit. on p. 7).

[18] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick, „Titanium Language Reference Manual, version 2.19“, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2005-15, 2005 (cit. on p. 7).

[19] K. Ebcioglu, V. Saraswat, and V. Sarkar, „X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access“, *International Workshop on Language Runtimes*, 2004 (cit. on p. 7).

[20] A. Skjellum, E. Lusk, and W. Gropp, „Using MPI: Portable Parallel Programming with the Message Passing Iinterface“, 1999 (cit. on p. 7).

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, „Cilk: An Efficient Multithreaded Runtime System“, *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995, ISSN: 0362-1340 (cit. on pp. 7, 93).

[22] F. Bodin and S. Bihan, „Heterogeneous Multicore Parallel Programming for Graphics Processing Units“, *Sci. Program.*, vol. 17, no. 4, pp. 325–336, 2009, ISSN: 1058-9244 (cit. on pp. 7, 94).

[23] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, „StarPU: a unified platform for task scheduling on heterogeneous multicore architectures“, Anglais, *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011 (cit. on pp. 7, 14, 37, 38, 54–58).

[24] W. Thies, M. Karczmarek, and S. Amarasinghe, „StreamIt: A Language for Streaming Applications“, in *International Conference on Compiler Construction*, Grenoble, France, 2002 (cit. on pp. 7, 94).

[25] C. Kessler, U. Dastgeer, S. Benkner, E. Marth, M. Sandrieser, J. L. Träff, and J. Singler, „PEPPHER Component Model, Deliverable D1.1 of the European Commission’s IST FP7 Project PEPPHER“, no. 10, 2010 (cit. on pp. 13–18, 25).

[26] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, „PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems“, *Micro, IEEE*, vol. 31, no. 5, pp. 28–41, 2011, ISSN: 0272-1732 (cit. on pp. 13, 14).

[27] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, A. Richards, R. Namyst, B. Bachmayer, C. Kessler, D. Moloney, and P. Sanders, „The PEPPHER Approach to Programmability and Performance Portability for Heterogeneous many-core Architectures“, English, in *ParCo*, Ghent, Belgique, 2011 (cit. on pp. 13, 15, 16).

[28] S. Benkner, S. Pllana, E. Marth, M. Sandrieser, C. Kessler, and U. Dastgeer, „PEPPHER Coordination Language and Component Composition Techniques, Deliverable D1.2 of the European Commission's IST FP7 Project PEPPHER“, no. 10, 2010 (cit. on pp. 13–17, 20–23, 25).

[29] U. Dastgeer, L. Li, C. Kessler, S. Benkner, E. Bajrovic, M. Sandrieser, and S. Pllana, „Research Prototype Implementation, Deliverable D1.4v2 of the European Commission's IST FP7 Project PEPPHER“, Project Report, 2012 (cit. on pp. 13, 37).

[30] S. Pllana and F. Xhafa, *Programming Multi-Core and Many-Core Computing Systems*. Chicester, GB: John Wiley and Sons Ltd, 2014, ch. PEPPHER: PERFORMANCE PORTABILITY AND PROGRAMMABILITY FOR HETEROGENEOUS MANY-CORE ARCHITECTURES, To appear, ISBN: 0470936908 (cit. on p. 14).

[31] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0201745720 (cit. on p. 14).

[32] G. Bradski and A. Kaehler, *Learning OpenCV*. O'Reilly Media Inc., 2008 (cit. on pp. 15, 19, 77).

[33] U. Dastgeer, L. Li, and C. Kessler, „The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-Based Systems“, in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012, pp. 711–720 (cit. on p. 22).

[34] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, „High-Level Support for Pipeline Parallelism on Many-Core Architectures“, in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Eds., vol. 7484, Springer Berlin Heidelberg, 2012, pp. 614–625, ISBN: 978-3-642-32819-0 (cit. on pp. 27, 30, 94).

[35] E. Bajrovic, S. Benkner, J. Dokulil, and M. Sandrieser, „Autotuning of Pattern Runtimes for Accelerated Parallel Systems“, in *PARCO 2013, September 2013, Munich, Germany*, 2013 (cit. on pp. 27, 96).

[36] John Watkinson, *The MPEG handbook: MPEG-1, MPEG-2, MPEG-4*, 2004 (cit. on p. 48).

[37] R. Namyst, C. Augonnet, S. Thibault, N. Furmento, U. Dolinsky, G. Russell, A. Richards, and D. Moloney, „Low-level Portability Layer, Deliverable D4.1 of the European Commission's IST FP7 Project PEPPHER“, no. 10, 2010 (cit. on p. 57).

[38] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi, „ROSE User Manual: A Tool for Building Source-to-Source Translators“, 2012 (cit. on p. 65).

[39] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811 (cit. on p. 66).

[40] „First the Tick, Now the Tock: Next Generation Intel® Microarchitecture (Nehalem)“, Intel Corporation, Tech. Rep., 2008 (cit. on p. 76).

[41] Nvidia Corporation, „NVIDIA's Next Generation CUDA Compute Architecture: Fermi“, Nvidia Corporation, Tech. Rep., 2009 (cit. on p. 76).

[42] „Intel® Xeon® Processor E5-1600/E5-2600/E5-4600 Product Families“, Intel Corporation, Tech. Rep., 2012 (cit. on p. 76).

[43] Nvidia Corporation, „NVIDIA's Next Generation CUDA Compute Architecture: KeplerTM GK110“, Nvidia Corporation, Tech. Rep., 2012 (cit. on p. 76).

[44] *Intel® Integrated Performance Primitives - User's Guide*, 2012 (cit. on p. 77).

[45] James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O'Reilly, 2007, pp. I–XXV, 1303–, ISBN: 978-0-596-51480-8 (cit. on p. 77).

[46] W. Kim and M. Voss, „Multicore Desktop Programming with Intel Threading Building Blocks“, *IEEE Software*, vol. 28, no. 1, pp. 23–31, 2011, ISSN: 0740-7459 (cit. on pp. 77, 93).

[47] P. Viola and M. Jones, „Rapid Object Detection using a Boosted Cascade of Simple Features“, in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, 2001, pages (cit. on p. 78).

[48] B. Bachmayer, S. Benkner, S. Pllana, E. Bajrovic, M. Sandrieser, U. Dolinsky, P. Keir, G. Russell, A. Richards, D. Moloney, and B. Barry, „PEPPHER Demonstrators, Deliverable D6.7 of the European Commission's IST FP7 Project PEPPHER“, 2012 (cit. on p. 85).

[49] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, „Rodinia: A Benchmark Suite for Heterogeneous Computing“, in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54 (cit. on p. 85).

[50] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, „A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads“, in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010, pp. 1–11 (cit. on p. 85).

[51] BLAS. (2014). Basic Linear Algebra Subprograms. BLAS - Netlib, [Online]. Available: `www.netlib.org/blas` (visited on 05/14/2014) (cit. on p. 93).

[52] W. Bosma, J. Cannon, and C. Playoust, „The Magma algebra system. I. The user language“, *J. Symbolic Comput.*, vol. 24, no. 3-4, pp. 235–265, 1997, Computational algebra and number theory (London, 1993), ISSN: 0747-7171 (cit. on p. 93).

[53] *PLASMA Users' Guide*, 2010 (cit. on p. 93).

[54] J. E. Stone, D. Gohara, and G. Shi, „OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems“, *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, 2010, ISSN: 0740-7475 (cit. on p. 93).

[55] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2 (cit. on p. 93).

[56] A. Benoit and Y. Robert, „Mapping pipeline skeletons onto heterogeneous platforms“, *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, pp. 790–808, 2008, ISSN: 0743-7315 (cit. on p. 93).

[57] Murray Cole, „Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming“, *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004, ISSN: 0167-8191 (cit. on p. 93).

[58] A. Pop and A. Cohen, „A Stream-computing Extension to OpenMP“, in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC ’11, Heraklion, Greece: ACM, 2011, pp. 5–14, ISBN: 978-1-4503-0241-8 (cit. on p. 93).

[59] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortí, „A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures“, in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, ser. IWOMP ’09, DresdenG, Germany: Springer-Verlag, 2009, pp. 154–167, ISBN: 978-3-642-02284-5 (cit. on p. 93).

[60] T. Preud’Homme, J. Sopena, G. Thomas, and B. Folliot, „An Improvement of OpenMP Pipeline Parallelism with the BatchQueue Algorithm“, in *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ser. ICPADS ’12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 348–355, ISBN: 978-0-7695-4903-3 (cit. on p. 93).

[61] C. Liao, Y. Yan, B. de Supinski, D. Quinlan, and B. Chapman, „Early experiences with the openmp accelerator model“, in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds., vol. 8122, Springer Berlin Heidelberg, 2013, pp. 84–98, ISBN: 978-3-642-40697-3 (cit. on p. 93).

[62] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, „Productive Programming of GPU Clusters with OmpSs“, in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 557–568 (cit. on p. 93).

[63] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, „An Extension of the StarSs Programming Model for Platforms with Multiple GPUs“, in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par ’09, Delft, The Netherlands: Springer-Verlag, 2009, pp. 851–862, ISBN: 978-3-642-03868-6 (cit. on p. 93).

[64] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer, „HyPHI - Task Based Hybrid Execution C++ Library for the Intel(R) Xeon Phi(TM) Coprocessor“, in *42nd International Conference on Parallel Processing (ICPP-2013), Lyon, France, 2013*, 2013 (cit. on p. 94).

[65] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer, „High-level Support for Hybrid Parallel Execution of C++ Applications Targeting Intel(R) Xeon Phi(TM) Coprocessors“, in *International Conference on Computational Science (ICCS 2013), Barcelona, Spain, June 2013. IEEE Computer Society Press.*, 2013 (cit. on p. 94).

[66] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and Beverly Bachmayer, „Efficient Hybrid Execution of C++ Applications using Intel(R) Xeon Phi(TM) Coprocessor. CoRR abs/1211.5530 (2012)“, Tech Report, 2012 (cit. on p. 94).

[67] Michael Wolfe, „Implementing the PGI Accelerator Model“, in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10, Pittsburgh, Pennsylvania: ACM, 2010, pp. 43–50, ISBN: 978-1-60558-935-0 (cit. on p. 94).

[68] OpenACC Standard. (2014). OpenACC - Directives for Accelerators, [Online]. Available: http://www.openacc-standard.org (visited on 05/19/2014) (cit. on p. 94).

[69] M. Aater, S. Moinuddin, K. Qureshi, K. Yale, and N. Patt, „Feedback-directed Pipeline Parallelism“, in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, Vienna, Austria: ACM, 2010, pp. 147–156, ISBN: 978-1-4503-0178-7 (cit. on p. 94).

[70] C. A. Schaefer, V. Pankratius, and W. F. Tichy, „Engineering Parallel Applications with Tunable Architectures“, in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, 2010, pp. 405–414 (cit. on p. 94).

[71] F. Otto, C. A. Schaefer, M. Dempe, and W. F. Tichy, „A Language-Based Tuning Mechanism for Task and Pipeline Parallelism“, English, in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. Guarracino, and D. Talia, Eds., vol. 6272, Springer Berlin Heidelberg, 2010, pp. 328–340, ISBN: 978-3-642-15290-0 (cit. on p. 94).

[72] B. Catanzaro and K. Keutzer, „Parallel Computing with Patterns and Frameworks“, *XRDS*, vol. 17, no. 1, pp. 22–27, 2010, ISSN: 1528-4972 (cit. on p. 95).

[73] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, „AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications“, in *Applied Parallel and Scientific Computing*, ser. Lecture Notes in Computer Science, P. Manninen and P. Öster, Eds., vol. 7782, Springer Berlin Heidelberg, 2013, pp. 328–342, ISBN: 978-3-642-36802-8 (cit. on p. 96).

[74] AutoTune Project. (2014). AutoTune Project's Official Website, [Online]. Available: http://www.autotune-project.eu (visited on 05/25/2014) (cit. on p. 96).

[75] L. Morin, E. Bajrovic, E. Cesar, M. Gerndt, C. Guillen, R. Miceli, C. Navarrete, A. Pimenta, and A. Sikora, „Design of the Tuning Plugins, Deliverable D4.1 of the Project AutoTune“, 2012 (cit. on p. 96).

[76] R. Miceli, C. Guillen, C. Navarrete, A. Pimenta, A. Sikora, L. Morin, H. Haitof, E. Bajrovic, F. Bodin, M. Sandrieser, and S. Benkner, „The Selected Applications including the Achieved Improvements by Manual Tuning and the Tuning Effort, Deliverable D5.1 of the Project AutoTune", 2012 (cit. on p. 96).

[77] R. Miceli, A. Pimenta, C. Navarrete, C. Guillen, H. Haitof, I. Comprés, E. Bajrovic, and L. Morin, „Test Results of the Integrated Prototype, Deliverable D5.3 of the Project AutoTune", 2013 (cit. on pp. 96, 97).

[78] S. Benkner and E. Bajrovic, „Automatic Performance Tuning of Pipeline Patterns for Heterogeneous Parallel Architectures", in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2014"*, to be published, 2014 (cit. on p. 96).

Curriculum Vitae

Name:	Enes Bajrovic
Date of birth:	25.07.1980

Education

1987 - 1991	Elementary School “Andrija Djurovic”, Titovo Užice, Serbia
1991 - 1995	Elementary School “Safvet Beg Bašagić”, Sarajevo, Bosnia and Herzegovina
1995 - 1999	Second Gymnasium Sarajevo, Bosnia and Herzegovina
2007 - 2011	Bachelor study in Scientific Computing, University of Vienna
2011 -	Master study in Scientific Computing, University of Vienna