Autonomic Behavioural Framework for Structural Parallelism over Heterogeneous Multi-Core Systems

Mehdi Goli

A thesis submitted in partial fulfilment of the
requirements of the
Robert Gordon University
for the degree of Doctor of Philosophy
May 2015

Abstract

With the continuous advancement in hardware technologies, significant research has been devoted to design and develop high-level parallel programming models that allow programmers to exploit the latest developments in heterogeneous multi-core/many-core architectures.

Structural programming paradigms propose a viable solution for efficiently programming modern heterogeneous multi-core architectures equipped with one or more programmable Graphics Processing Units (GPUs). Applying structured programming paradigms, it is possible to subdivide a system into building blocks (modules, skids or components) that can be independently created and then used in different systems to derive multiple functionalities.

Exploiting such systematic divisions, it is possible to address extra-functional features such as application performance, portability and resource utilisations from the component level in heterogeneous multi-core architecture. While the computing function of a building block can vary for different applications, the behaviour (semantic) of the block remains intact. Therefore, by understanding the behaviour of building blocks and their structural compositions in parallel patterns, the process of constructing and coordinating a structured application can be automated.

In this thesis we have proposed Structural Composition and Interaction Protocol (SKIP) as a systematic methodology to exploit the structural programming paradigm (Building block approach in this case) for constructing a structured application and extracting/injecting information from/to the structured application. Using SKIP methodology, we have designed and developed Performance Enhancement Infrastructure (PEI) as a SKIP compliant autonomic behavioural framework to automatically coordinate structured parallel applications based on the extracted extra-functional properties related to the parallel computation patterns.

We have used 15 different PEI-based applications (from large scale applications with heavy input workload that take hours to execute to small-scale applications which take seconds to execute) to evaluate PEI in terms of overhead and performance improvements. The experiments have been carried out on 3 different Heterogeneous (CPU/GPU) multi-core architectures (including one cluster machine with 4 symmetric nodes with one GPU per node and 2 single machines with one GPU per machine). Our results demonstrate that with less than 3% overhead, we can achieve up to one order of magnitude speed-up when using PEI for enhancing application performance.

To my beloved, friend and guide,

Chariji

Acknowledgements

First and foremost, I would like to thank my father who was my main motivator in perusing my education to the highest degree.

I would like to thank my wife who has been with me through all difficult periods that I went through during these years and her advice for handling those difficult situations.

I would like to thank my family as well who patiently tolerated the hard situation of minimum contact and support from me through these years.

I would like to thank my supervisory team for their efficient advice and support, both morally and technically, without which it would not have been possible for me to complete the PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Mehdi Goli*)

Nomenclature

ACG Abstract Computation Graph

AMD Advanced Micro Devices

API Application Program Interface

APUs Accelerated Processing Units

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DSL Domain Specific Language

DSRI Dynamic Skeleton Runtime Interface

DSP Digital signal processing

FF FastFLow

FIFO First-in First-out

FP Functional Programming

FPGA Field-Programmable Gate Array

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

GS Group Size

HAL High-Level Abstraction Layer

HFastFlow Heterogeneous FastFLow

Intel TBB Intel Threading Building Block

LDS Local Data Share

MCTS Monte Carlo Tree Search

MD Molecular Dynamic

MISD Multiple Instruction Single Data

MPI Message Passing Interface

ODVL OpenCL Device Virtualisation Layer

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PE Processing Element

PEI Performance Enhancement Infrastructure

PETs Performance Enhancement Tools

POSIX Portable Operating System Interface

RISC-Pb^{2}l Reduced Instruction Set Computing based Parallel Building Block Library

RPC Remote Procedure Call

SIMD Single Instruction Multiple Data

SKIP Structural Composition and Interaction Protocol

SMs Streaming Multi-Processors

SMTWTP Single Machine Total Weighted Tardiness Problem

UCT Upper Confidence bounds applied to Trees

UML Unified Modelling Language

URNG Uniform Random Number Generator

ZMQ Zero Message passing Queue

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Before 1990, there were very few parallel computers, which were only used for the most critical problems. However, the availability of parallel computers has changed dramatically since the mid 1990s. With the appearance of multiple processor cores and multi threading support, parallel computers are becoming ubiquitous. Now, almost all university computer science departments have at least one parallel computer. Virtually all oil companies, auto mobile manufacturers, drug development companies and special effects studios use parallel computing.

Although computing in less time is beneficial, and may enable problems to be solved that cannot be otherwise, it comes at a cost. Writing software to run on parallel computers can be difficult. Only a small minority of programmers have experience with parallel programming. If all these computers that are designed to exploit parallelism are going to achieve their potential, more programmers need to learn how to write parallel programmes. They need to deal with more complexities such as synchronising multiple processes/threads, handling communication between processes/threads, preventing deadlocks, and, even more, after all of these, programmers need to think about optimisation in order to achieve the best performance. Otherwise, the result might be even worse than running a programme in sequential mode.

Recent revolutions in underlying hardware have also had an important effect on parallel programming algorithms. With the appearance of the general purpose graphical processing unit (GPGPU), a new parallel programming environment is required to support the execution of the programme in a hybrid (multi-core CPU/GPGPU) environment. Several environments such as CUDA C and OpenCL are available, allowing the programmer to develop applications which can take advantage of both CPUs and GPUs.

One the one hand, GPUs have been shown to have orders of magnitude faster than CPUs for different application domains. In [1] there is an order of magnitude speed-up for a computationally intensive portion of the Weather Research and Forecast (WRF) model on a variety of NVIDIA Graphics Processing Units (GPU). In [2] with the use of a GPU accelerator the k nearest neighbor search (KNN) algorithm has been speed-up for two orders of magnitude.

In [3], one order of magnitude speed-up has been achieved for the GPU-based implementation of the video feature tracking and matching technique.

In [4], an image correlation as a fundamental component of many real-time imaging and tracking systems has been implemented on a graphics processing unit (GPU) with one order of magnitude speed-up over the CPU version.

On the other hand, GPUs make the complexities of parallel programming even harder than before. A programmer needs to take care of the following:

- Using different parallelisation techniques such as tiling to achieve maximum reasonable speed-up and resource utilisation.
- Managing the usage of shared memories registers, global memories, constant memories and providing coalesce memory access to hide the memory latency, while in multi-core systems such optimisations are automatically handled by compilers.
- Minimising the data transfer cost between GPU memory and CPU memory to minimise the I/O time and reduce the processing elements' idle time.

Things can be even more complex if we have more than one GPU-device due to the need for extra efforts such as the coordination of tasks and distribution of data among GPUs based on their memory limitations, the speed of processing elements, and I/O bus speed between GPUs and CPUs.

One of the major problems in parallel programming is the lack of abstraction. The CUDA and OpenCL environments supporting hybrid (CPU and GPU) executions are low-level and they do not provide a high-level abstraction for the programmer to hide the above-mentioned complexities. Several high-level approaches are offered to tackle the abstraction issues by providing high-level techniques such as compiler based directive languages [5, 6, 7] or structured parallel programming [8, 9, 10, 11].

Although compiler based directive languages are providing an abstraction mechanism, they are highly inflexible, both in terms of not allowing dynamic adaptation in their execution environment, and in terms of making it difficult to introduce the high-level changes to programme structure.

As a well-known example of structured programming, *algorithmic skeletons* have long been considered a viable way of introducing high-level abstraction into parallel programming that hides the complexity of recurring patterns of coordination and communication logic behind a generic reusable application interface [12, 13].

Moreover, the building block based approach is another example of structured programming that supports the modelling and implementation of high-level structured parallel programming frameworks. It has been demonstrated that such approach can not only support

skeleton based structured programming, but also general purpose programming models and domain specific patterns [14, 15].

Beside synchronisations and communications, another coordination problem can be optimising the scheduling mechanism in hybrid environments. Several researchers have demonstrated that operating systems provide a robust scheduling mechanism to support multi-threading and separate the coordination of allocating different threads to different resources from the programmer [16, 17, 18, 19, 20]. However, executing applications over heterogeneous multi-core/many-core resources requires an explicit scheduling system in order to optimise performance and resource utilisation.

In [21] an extension of the OpenMP approach has been provided that spreads the compiler provided parallel computation across the heterogeneous cores to optimize performance and power consumption. In [22] a uniform interface is designed for task schedulers, offering low level scheduling mechanisms such as work stealing. Depending on the application, a user can select the most appropriate strategy at runtime, as all strategies implement the same interface.

In [23, 24] a scheduling approach has been designed which allocates incoming streams to a set of available CPUs and GPUs with a determined order to guarantee that no deadlines will be missed. In [25, 26] two different scheduling algorithms have been provided to extend the execution of an OpenCL programme written on a single device on both CPU and GPU in order to fully utilise all available OpenCL-enabled devices in a system. In [27] automatic adaptive mapping has been developed to dynamically map computation units on processing elements executing on both CPU and GPU.

Such a scheduling system can be considered as glue between the device scheduler and the operating system scheduler to optimise the allocation patterns of heterogeneous applications. Several approaches have emerged to tackle the problem of scheduling over heterogeneous multi-core architectures such as heuristic approaches, source-to-source compiler based optimisation and historical information generated from previous executions [28, 29, 30, 31].

Moreover, *autonomic management* can be considered as a viable solution to overcome the rapidly growing complexity of computing systems management and to reduce the barrier that complexity poses to further growth. **Autonomic management** refers to the self-managing characteristics of computing resources and adapting to unpredictable changes while hiding intrinsic complexity to operators and users[32].

By applying **autonomic management** in the parallel computing area, the self-managing characteristics can be considered as extra-functional and non-functional features such as performance, portability, security, fault-tolerance and power-management. In this case, applying autonomic management on a skeleton based framework to control a set of extra-functional and non-functional features has been called *behavioural skeletons*. Therefore, a **behavioural skeleton** is a skeleton with an associated autonomic manager taking care of extra-functional

and non-functional properties related to skeleton implementation. It can be constructed as a combination of parallel patterns and rule-based control systems. Such a system delivers the implementation of an autonomic mechanism while managing one or more extra-functional or non-functional properties related to the parallel computation patterns [33].

In [34, 35, 36, 37] different behavioural modelling frameworks have been introduced on grid systems for autonomic management where the quality of a service goal can be defined for throughput under varying conditions of resource availability.

In [38], an investigation into key technologies has been performed to provide a new paradigm for developing autonomic grid applications. The proposed paradigm can be used to develop applications that are context aware and capable of self-configuring, self-composing, self-optimising and self-adapting.

In [39], an extension of the Fractal component model has been proposed to allow the modular development of adaptation policies and their dynamic weaving into running applications. These policies detect the evolutions of the execution context and adapt the base programme by reconfiguring it.

In principle, considering the abstraction and autonomic coordination over a set of extra-functional features as two of the major challenges in the parallel computing area, it is expected that an efficient parallel programming model:

- Should provide a reasonable level of abstraction to hide the difficulty of communication and synchronisation over heterogeneous multi-core environments; and
- Must provide an autonomic management to optimise execution of parallel applications on a set of available heterogeneous resources over a set of predefined extra-functional features.

The challenge is, therefore, to produce and support such a programming paradigm that satisfies the above two conditions.

Although different behavioural skeletons automate the coordination of heterogeneous parallelism, they typically support non-functional properties, such as quality of service in grid computing systems. To the best of our knowledge, the autonomic management approach has not been applied to heterogeneous multi-core architecture for both extra-functional and non-functional properties.

Moreover, the existing autonomic management approaches for parallel programming have mainly been applied to algorithmic skeleton and have not considered other structured programming models. In [14, 15] it has been demonstrated that the intrinsic characteristics of structured parallelism through the building block approach place this paradigm in a preponderant position to support a reasonable level of abstraction by implicitly providing the communication and synchronisation for a parallel application. Also, it has been demonstrated that this approach

supports not only common patterns in algorithmic skeletons, but also other approaches such as the Google Map-Reduce model and domain specific parallel programming models. Therefore, our behavioural framework uses RISC-Pb^{2}l building block approach presented in [14, 15] as a structured parallel programming approach to support abstraction.

Thus, in this thesis we develop a behavioural RISC-Pb^{2}l framework that supports autonomic management over heterogeneous multi-core architecture for both extra-functional and non-functional properties.

To develop the autonomic behavioural framework over heterogeneous multi-core architecture, this thesis presents SKIP as a generic strategy that targets the autonomic coordination on structural parallelism over heterogeneous multi-core architectures. Applying the SKIP methodology, it would be possible to turn an existing structural parallel framework into a SKIP compliant autonomic behavioural framework with minimum modifications. Therefore, it would be possible for such a framework to interact with a set of SKIP compliant coordination engines to support autonomic coordination over heterogeneous multi-core architectures.

> SKIP (*Structural Composition and Interaction Protocol*) is a generic methodology to instrument the structural programming for further autonomic coordination over one or more aspects of extra-functional properties related to parallel computation patterns on heterogeneous multi-core architectures.

Exploiting the SKIP interaction mechanism, a SKIP-compliant behavioural framework can automatically detect any variation regarding the selected extra-functional or non-functional properties and enhance it. Such variation can be a change in data input size, application priority or termination of a component in an application. Once such variation is detected, the automatic coordination is applied through a set of SKIP-compliant coordination engines to optimise the performance.

1.1 Contributions

In this thesis we have designed and implemented an autonomic behavioural framework called performance enhancement infrastructure (PEI) supporting automatic coordination management for structured parallel applications over heterogeneous multi-core architectures.

The framework is composed of the following three main parts:

1. **Structured Parallel Programming Model**: Separates coordination from computation and hides the synchronisation and communications between application components from end-users.

2. **High-Level Abstraction Layer (HAL)** supports autonomic management for structured applications over heterogeneous multi-core architectures;

3. **Performance Enhancement Tools (PETs)** which are a set of coordination engines that support scheduling, load balancing and static configuration of an application structure for a specific heterogeneous multi-core architecture. We have designed and developed these coordination engines as part of the key optimisation objectives for ParaPhrase project [40].

Structured Parallel Programming Model: We have selected the existing RISC-Pb^{2}l building block approach as the structured parallel programming model. We have:

- extended the RISC-Pb^{2}l by adding a new heterogeneous block that enables the coordination of application over heterogeneous multi-core architecture;
- designed a grammar for the extended RISC-Pb^{2}l that detects the set of parallel programming patterns supportable by RISC-Pb^{2}l building block library for heterogeneous multi-core architecture. The grammar checks the correction and validation of the application structure generated by RISC-Pb^{2}l library;
- used FastFlow [41] as an existing RISC-Pb^{2}l framework for homogeneous multi-core system; and,
- expanded FastFlow by implementing a GPU back-end that supports application execution over heterogeneous multi-core architectures. This expansion is the implementation of the new heterogeneous building block called HFastFlow.

High-Level Abstraction Layer (HAL): Engulfing a structured parallel programming framework, this layer delivers three interfaces:

1. *User-level interface*: Determines a set of extra-functional and non-functional properties that allows the end-user to descriptively define the structural composition of its application. It also allows the end-user to descriptively determine the constraint configuration of the application and the underlying architecture.
2. *System level interface*: Determines a set of extra-functional and non-functional properties that are *i)* monitored on the structured parallel framework; *ii)* monitored on the underlying resources; *iii)* addressed by coordination engines.
3. *Autonomic manager interface*: Is a bridge between the end-user, the structured framework, the coordination engines, and the underlying hardware. It can automatically apply the specific coordination decisions to adapt to the changes happening on both the application status or the underlying hardware status.

In order to develop HAL, we have:

- determined a set of controlling parameters that can affect application coordination. This has been provided by investigating a set of existing structured parallelism framework that support coordination over heterogeneous architectures;
- designed SKIP as a generic methodology that enables autonomic management for structured parallel programming model over heterogeneous multi-core architectures. SKIP determines both user-level and system level interfaces required to provide an autonomic management over a structured parallel application;
- fused SKIP with the RISC-Pb^{2}l grammar to generate the high-level abstraction layer for RISC-Pb^{2}l building block approach;
- instrumented HFastFlow by adding a set of actuators and sensors to exchange the extra-functional and non-functional properties determined in SKIP. This instrumentation represents the implementation of the SKIP fusion on RISC-Pb^{2}l;
- designed and implemented a SKIP adaptor that auto-generates a RISC-Pb2l application from a descriptive structural composition file provided by the end-user; and,
- designed and implemented a dynamic skeleton runtime interface (DSRI) to provide the autonomic management on HFastFlow for heterogeneous multi-core architecture.

Performance Enhancement Tools (PETs): We have designed and implemented a set of coordination engines in order to verify the suitability of HAL for orchestrating structured parallel applications over heterogeneous architecture. These engines interact with HFastFlow through the autonomic manager (DSRI) to optimise performance of the application and its resource utilisation over heterogeneous architectures. The coordination engines we have designed and implemented in this thesis are as follows.

- A scheduling mechanism to allocate/reallocate different components of applications on the underlying heterogeneous devices. It can dynamically remap heterogeneous software components to the available CPU/GPU devices based on the following information provided by the high-level abstraction layer: *i)* the extra-functional properties of the software components; *ii)* the hardware performance characteristics; and *iii)* information that is obtained from monitoring the dynamic system load. The provided system is capable of dealing with components from multiple applications and remapping them to the best available hardware, thus ensuring optimal use of the available hardware resources.
- A load-balancing technique that auto-tunes the workload fraction over different heterogeneous components with regards to their computational power. The technique

will also respond to sudden changes on the input workload size and if required, automatically tunes the workload fraction for application components.

- A static structural configuration technique to tune an application structure for a specific heterogeneous architecture. As an external coordination method, the static structural configuration tries to automatically tune a structured application and to map its components to the available resources for a given architecture in order to maximise the application throughput.
- An efficient idling technique to enhance utilisation of the resources for any CPU slot allocated to a component. The efficient utilisation of a resource depends on the availability of a task in a component queue for the allocated slot for the component.
- A memory management technique that dynamically controls the memory usage for heavy workload applications in order to prevent queue overflows for application components in HFastFlow.

PEI Evaluation: In order to verify the efficiency of PEI, we have used 15 high-level structured applications classified into *two categories*.

The first category consists of 6 different *image processing applications* using the OpenCL back-end that we have designed and implemented in PEI. These applications are presented in Chapter 5, section 5.1. Each application in this category receives a stream of 6144 images as the input to process. This category has been designed to verify the efficiency of coordination engines regarding the structural extension to the FastFlow framework for targeting heterogeneous multi-core architectures. These applications have been executed on 2 different heterogeneous multi-core architectures, operating on different workload varieties from small-scale data to relatively large-scale data.The result are:

- less than 3% performance overhead for using PEI to enhance the performance of image processing applications;
- up to one order of magnitude speed-up over serial CPU version when OpenCL back-end is applied;
- up to 1.68 times speed-up over serial GPU version when OpenCL back-end is applied;
- up to 38% increase in GPU utilisation;
- up to 9 times speed-up when using the adaptive load-balancer instead of the FastFlow round-robin load-balancer;
- up to one order of magnitude speed-up when applying our dynamic scheduling over the FastFlow round robin technique;
- up to 5 times speed up when applying the static structural configuration; and

- no drop in performance when executing multiple applications concurrently.

The second category consists of 9 *existing FastFLow applications that do not use the extended OpenCL back-end*. This category has been designed to attest the efficiency of the proposed approach over a set of existing applications in the FastFlow framework.

This category includes an *N*-body simulation, 5 standard FastFlow benchmarks the EISPACK routine, a Molecular Dynamic (MD), and SMPWTP application. All applications are comprehensively described in Chapter 6. The results are:

- less than 3% performance overhead when using PEI to enhance the performance of FastFlow benchmark applications;
- up to 2 times speed-up more than Thrust and SkePU for the *N*-body simulation on large input data sizes;
- up to 2 times speed-up when using our efficient idling technique for FastFlow benchmark applications;
- up to 20 times speed-up when applying static structural configuration for molecular dynamic on Titanic machine;
- up to 5 times speed-up when applying static structural configuration for SMTWTP application on Titanic machine; and,
- Efficiently controlling the memory usage for EISPACK a known large-scale application with a heavy input workload.

1.2 List of Publications and Authorship

This research has been supported by the European Commission under the 7^{th} Framework Programme through the Specific Targeted Research Project 'ParaPhrase': Parallel Patterns for Adaptive Heterogeneous Multicore Systems" http://paraphrase-ict.eu/, contract no.: 288570. The ParaPhrase project has produced a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component based applications that can be mapped to the available hardware resources and, if required, dynamically re-mapped to meet application needs and hardware availability.

Some of the results of this research have been consistently published in high-impact outlets. For each publication below, we briefly explain the main contribution of the paper and the relationship to this thesis.

[1] Mehdi Goli, Horacio Gonzalez-Velez: N-body computations using skeletal frameworks on multicore CPU/graphics processing unit architectures: an empirical performance

evaluation. Concurrency and Computation: Practice and Experience 26(4): 972-986, 2014.

In this paper we have developed *N*-body-simulation applications on three different structured frameworks, namely, FastFlow, Thrust, SKePU. We have compared the performance overhead and speed-up of *N*-body-simulation for 1024, 8192 and 65535 bodies on these three frameworks. The results have been added to chapter 6.

[2] Mehdi Goli, Michael T. Garba, Horacio Gonzalez-Velez: Streaming Dynamic Coarse-Grained CPU/GPU Workloads with Heterogeneous Pipelines in FastFlow. HPCC-ICESS. 445-452, 2012.

In this paper we have investigated the generality and scalability of FastFlow to parallelise an existing application called *EI*-routine. Moreover, we have efficiently controlled the memory usage for the EISPACK application as a large-scale application with a heavy input workload. The result for the memory management technique has been added to chapter 6. We have also determined the necessity of GPU-back-end support in FastFlow to further coordinate an application over heterogeneous architecture.

[3] Mehdi Goli, Horacio Gonzalez-Velez: Heterogeneous Algorithmic Skeletons for Fast Flow with Seamless Coordination over Hybrid Architectures. PDP 148-156, 2013.

In this paper we have expanded FastFlow by implementing an OpenCL-based GPU back-end that supports application execution over heterogeneous multi-core architecture. This expansion has been added to chapter 4, while the implementation of the new heterogeneous building block is presented in chapter 3.

[4] Mehdi Goli, John McCall, Horacio Gonzalez-Velez, and Marco Aldinucci. Performance Enhancement Infrastructure for Skeleton-based Parallel Programming Frameworks In CSRD Springer, submitted, 2014.

In this paper we have implemented the PEI framework and evaluated its overhead and efficiency to coordinate a set of FastFlow benchmark application. The PEI implementation has been added in chapter 4 and its evaluation has been added to chapter 6.

[5] Mehdi Goli, John McCall, Christopher Brown, Vladimir Janjic, Kevin Hammond: Mapping parallel programmes to heterogeneous CPU/GPU architectures using a Monte Carlo Tree Search. IEEE Congress on Evolutionary Computation 2013: 2932-2939.

Our contribution to this paper was to develop a static-structural configuration technique to tune an application structure for a specific heterogeneous architecture. Furthermore, for the specified architecture, the algorithm statically maps application components to the available CPUs/GPUs. The static structural configuration technique is a Monte-Carlo Tree Search (MCTS)-based algorithm which has been added to chapter 4

[6] Christopher Brown, Vladimir Janjic, Kevin Hammond, Kamran Idrees, Colin Glass, Amer Wafai, Mehdi Goli, and John McCall. Bridgind the divide: a new methodology for semi-automatic programming of heterogeneous parallel machines. In CSRD Springer, submitted, 2014.

In this paper we have integrated our MCTS-algorithm with the refactoring technique provided by other Paraphrase partners to automate the process of static skeleton trimming and component mapping for an application generated by the refactoring tool. We have evaluated our MCTS-approach over two parallel applications (namely molecular dynamic and SMTWTP) generated by the refactoring tool. The evaluation results have been added to chapter 6.

1.3 Research Method

The discipline of computer science has roots in three other disciplines: mathematics, natural science and engineering. Theory, abstraction and design are three paradigms proposed by The ACM Task Force on the core of computer science [42] to characterise research work in computer science.

Theory originates from mathematics where the objects of study are identified, and their relationships are hypothesised, proved and later interpreted. Abstraction is ingrained in experimental scientific methods where a hypothesis is formed, models are built to predict and in the end, experiments are carried out to evaluate built models. Finally, the results are analysed. Design has its essence in engineering where requirements are specified, and a solution is designed, built and tested to ensure that it satisfies the given requirements.

As pointed out in [42], all three paradigm are equally important and are often intertwined together in computer science. Moreover, all three paradigms model a process which can be repeated and iterated multiple times until the desired outcome is achieved. The research presented in this thesis mainly follows the abstraction and design paradigms.

The proposed autonomic behavioural model represents an abstract paradigms that instruments structural parallel applications for further autonomic coordination over one or more aspects of more extra-functional properties of parallel computation patterns on heterogeneous multi-core architectures. The design and development of the behavioural model followed by different evaluations regarding the feasibility and efficiency of the framework that represents the design paradigm.

1.4 Thesis Architecture

This thesis has been organised into seven chapters and four appendices. In the following section we provide a brief description of each chapter.

1. **The Introduction** presents an overall description of our approach by introducing structured parallelism as a way to employ autonomic interaction with structured applications and states the hypothesis, followed by the enumeration of the objectives and structure of the thesis.

2. **The Review of Literature** presents an outline of the structured parallelism paradigm and its related approaches, with strong emphasis on heterogeneous systems in order to put the contribution of this thesis into context. This chapter ends with the identification of the specific research gap addressed by this work.

3. **The SKIP Methodology** This chapter presents the SKIP Methodology for designing a behavioural system that supports autonomic management (including monitoring and coordination processes of instrumented components) on RISC-Pb2l-applications during their lifecycles. Through this chapter we have:

   (a) expanded the RISC-Pb^{2}l building block approach to support heterogeneous architecture;

   (b) presented a grammar for RISC-Pb^{2}l building block approach on heterogeneous architecture;

   (c) introduced SKIP to enable autonomic management for structured parallel programming model over heterogeneous multi- core architectures;

   (d) fused SKIP with the provided RISC-Pb^{2}l grammar to instrument the building block approach for structural constructing and interacting with RISC-Pb^{2}l-applications.

   (e) designed a SKIP compliant behavioural system to support autonomic coordination for RISC-Pb2l-applications on heterogeneous multi-core architectures.

4. **The Implementation of the SKIP-Compliant Behavioural Framework** In this chapter we have proposed PEI as the implementation of SKIP-compliant behavioural system that is proposed by the SKIP methodology. This chapter is composed of four different parts, namely:

   i FastFlow expansion (GPU back-end) to support heterogeneous architecture;

   ii FastFlow instrumentation with actuators and sensors;

   iii Implementation of HAL which support autonomic management over HFastFlow framework; and,

iv Implementation of coordination engines that deliver autonomic adjustment related to the determined extra-functional properties.

5. **The Evaluation of OpenCL Based Applications** represents the evaluation of the SKIP-compliant behavioural framework on a set of applications using our structural expansion on FastFlow over heterogeneous systems. It contains the application definition, assessment and discussion about the applicability of different coordination methods targeting different extra-functional properties.

6. **The Evaluation of Generic Applications** represents the evaluation of the SKIP-compliant behavioural framework on a set of existing FastFlow applications which does not use our structural expansion for FastFlow over heterogeneous systems. It contains the application definition, assessment and discussion about the applicability of some coordination methods that are suitable for those applications.

7. **Conclusion & Future Work** The conclusion furnishes a description of the research area covered by the thesis. It explains the ongoing related research studies regarding our investigation. It concludes by discussing possible avenues for future work.

8. **Appendix: Validation of RISC-pb^{2}l Grammar** This appendix chapter explains the validation of the provided RISC-pb^{2}l grammar for the set of supported patterns presented.

9. **Appendix: The Structural Representation of the Application Suite** This appendix chapter demonstrates the structural representation of designed applications through the RISC-pb^{2}l grammar and SKIP methodology.

10. **Appendix: The SKIP Compliant Objects** This appendix chapter indicates a snapshot of the interaction mechanism provided by the SKIP methodology for the different evaluated applications.

11. **Appendix: The Structural Representation of Existing Applications** Using the provided RISC-pb^{2}l grammar, this appendix chapter demonstrates the structural representation of existing applications used for evaluation.

12. **Appendix: Implementation of *N*-body Simulation Under Three Frameworks** This appendix chapter indicates the implementation of the *N*-body simulation for three different frameworks: FastFlow, Thrust and SKePU.

Chapter 2

Review of Literature

This thesis presents an empirical study of the autonomic coordination of parallel applications over heterogeneous multi-core architectures based on the use of structured parallelism foreknowledge.

In this chapter first, we explain some expressions and concepts used in this research. Then, we provide an outline of relevant concepts followed by a review of the current literature related to this subject and set the concept of high level abstraction and optimisation in structured parallel programming applications with a strong emphasis on heterogeneous systems. The wide range of research in the parallel programming domain is dedicated to the two following subjects:

Providing an abstraction layer that hides the complexity of communication, coordination, synchronisations and allocation on different architectures; and,

Providing an optimisation mechanism to (auto-)tune the performance of parallel applications.

Analysing the scant research regarding the two subjects, the former has been reviewed in section 2.2, while the latter has been reviewed in section 2.3 Finally, section 2.4 indicates the specific research gap addressed by this thesis.

2.1 Preliminaries

In this section we explain some expressions and concepts used in this research.

The following terms are employed here which may have been used differently in other literatures.

- Node: Refers to an independent unit of a programme that can be executed on a device.
- Processing element (PE): Refers to the processing element on which each node is executed. For heterogeneous computations, the granularity of a processing element is CPU

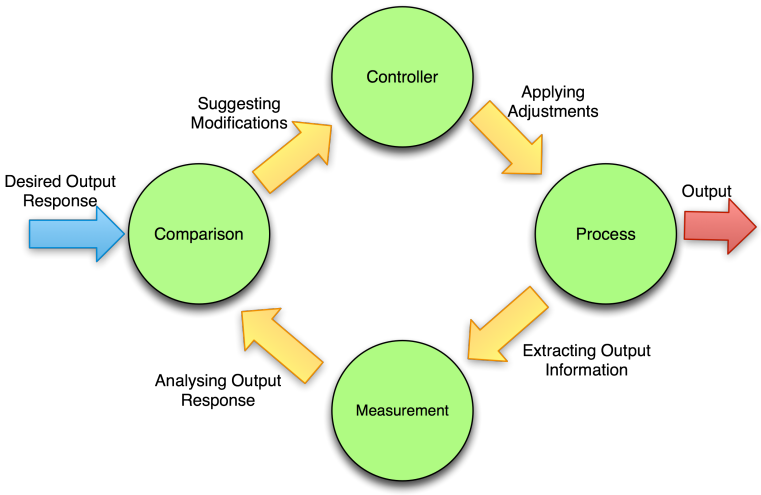


Figure 2.1: Feedback control system

or GPU. Although each of these is composed of different smaller elements such as cores in CPUs and streaming multi-processors(SM) in GPUs, the decision about the further assignment of nodes to the core of a CPU or SMs in a GPU is handled by the scheduling system of each PE. For CPUs this is handled by an operating system scheduler and for GPUs the device scheduler is responsible for further allocations.

- Task: Is the unit of data executed by each node in each iteration.

2.1.1 Control Systems

Control Systems can be defined as a set of interconnected components operating with each other to maintain an actual system performance that is close to a desired set of performance specifications. Such interconnections form system configurations that provide a desired system response as time progresses [43].

Figure2.1 demonstrates a feedback control system. It senses the changes through sensors or measurement units in the output due to the process changes via comparison units and attempts to correct the output through actuators or controller units. The sensitivity of a control system to parameter variations is an important factor that is controlled by damping ratios.

2.1.2 OpenCL

OpenCL (Open Computing Language) is an open, royalty-free and low-level standard provided by the Khronos group [44, 45]. It offers a unified computing platform for modern heterogeneous systems. Different vendors such as NVIDIA, AMD, Apple and Intel have released the realisation of the OpenCL, mainly targeting their own computing architectures. All OpenCL vendors provide a C-like programming language where developers can develop their compute

kernels. An OpenCL code is compiled dynamically by calling the OpenCL API functions. The dynamic compilation provides an optimum utilization of the latest software for underlying devices and hardware features such as SIMD computing capability. The OpenCL code is automatically uploaded to the OpenCL device memory at the fist invocation.

In the OpenCL terminology, a programme executes on an OpenCL-capable device (CPU, GPU, FPGA, etc.) which contains compute units (one or more cores) and may include one or more single instruction multiple data (SIMD) processing elements. The OpenCL not only hides the threads, but also abstracts hardware architectures and provides a common parallel programming interface. It provides a programming environment composed of a host CPU and connected OpenCL capable devices which may or may not share memory with the host and might have different machine instruction sets.

In OpenCL, a *work-item* is a single element of a problem domain. Grouping a set of work-items together will form a *work-group*. OpenCL represents four types of memories including [46]:

Global memory: All work-items in all work-groups have read/write access to the global memory. However, using the global memory has high access latency and therefore access to the global memory should be minimal.

Constant memory: Being part of the global memory, constant memory retains its constant values throughout the kernel execution.

Local memory: Provides shared variable spaces for all work-items of a work-group with read/write access.

Private memory: Provides private spaces that are visible to each individual work-item for read/write data access.

The OpenCL local and private memories on GPU devices are in an on-chip local data share (LDS) and private memory bank with each computing unit respectively. CPU devices implement the private memory as a register/L1 cache, local memory as a L2/L3 cache, global as a main memory, and constant OpenCL memory as a main memory/cache. However, their exact implementations are architecture dependent. The execution starts on the host programme, which manages one or more OpenCL devices by enqueuing the kernel-execution and memory-transfer commands to the devices' command queues.

In principle, OpenCL programmes should be portable (executable) on all OpenCL platforms (e.g., x86 CPUs, DSPs, AMD and NVIDIA GPUs). However, in reality, certain modifications may be required for switching between different OpenCL implementations [47]. Moreover, device-specific optimisations applied to an OpenCL application may have a negative performance impact when porting the application from one type of OpenCL device to another [47, 48].

2.2 Abstraction Mechanism

One of the major problems in parallel programming is the lack of abstraction. Several approaches are used to tackle the abstraction problem for developing parallel applications which can be mainly divided into two categories:

- Traditional low-level library model
- Structured high-level parallel programming model

2.2.1 Traditional Low-level Library Model

The traditional approach was to provide compiler based directive language low-level libraries, such as OpenMP, MPI or POSIX threads to introduce parallelism on multi-core CPUs. Such approaches are widely used in non-numerical applications.

With the emergence of heterogeneous GPGPUs in multi-core environments, several OpenMP directive approaches have been provided to hide the difficulties from users. An example of these approaches is PGI [5], HMPP [6] and OpenACC [7]. In all cases the programmer uses compiler directives to mark regions of code that need to be executed on the GPU. Then, the compiler uses these directives and generates the executable code targeting the GPU. It also allocates the required memory on the GPU and transfers the data from the host to the GPU and vice versa. The programmer needs to explicitly specify these operations by using appropriate compiler directives.

However, such approaches are highly inflexible, both in terms of not allowing dynamic adaptation to their execution environment, and in terms of making it hard to introduce the high-level changes to a programme structure that may be necessary to support new multi-core computer architectures, or to refactor an existing application to support a new parallel application.

2.2.2 Structured High-level Parallel Programming Model

Structured parallel programming is defined as the composition of nestable parallel patterns that avoide additional dependencies for both data and control flows [49]. There are several approaches to providing a high-level structured parallel application which can be defined as follows.

Algorithmic skeletons

As introduced by Cole [12], *algorithmic skeletons* abstract commonly-used patterns of parallel computation, communication and interaction. Skeletons demonstrate a top-down structured

approach, where parallel programmes are formed from the parametrisation of nested skeletons, which is also known as structured parallelism [13].

Deployed through skeleton frameworks, structured parallelism yields a clear and consistent structure across platforms by distinctly decoupling computations from the structure in a parallel programme. It is independent from specific hardware and it benefits entirely from any performance improvements in the system infrastructure. Demonstrated as either new languages or libraries, algorithmic skeleton frameworks can benefit from a variety of techniques including macro-data flow, templates, aspect-oriented programming and rewriting techniques to target distributed architectures (e.g. grids) and, more recently, multi-core architectures [50].

Although, algorithmic skeletons (and indeed parallel patterns in general) are not universally applicable to the production of parallel and distributed programmes, there are a significant number of applications that contemplate them as a relevant research. As they are derived and inherited from algorithmic skeletons, the well-known *Google MapReduce* is an example of a recent and highly successful "programming model". Furthermore, capturing the structure of the programme, skeletal methodologies inherently possess a predictable communication and computation structure. They provide, through construction, a foundation for performance modelling and an estimation of parallel applications [50, 51].

Parallel Patterns

Parallel patterns are high-level parallel programming models for parallel and distributed computing. As proposed in [52, 53] parallel design patterns can be considered as a potential approach to inducing a radical change in simplifying parallel programming. Abstracting the communication/computation mechanism, parallel design patterns can be useful for educating a new generation of parallel programmers to manage the new parallel architectures provided by hardware vendors [54]. Table 2.1 indicates examples of well-known patterns [55].

Building Block Based Approach

The building block based approach attempts to decompose parallel patterns and skeletons into a set of essential building blocks where each block can execute independently from the other blocks. Depending on its defined semantic, each building block wraps a unit of computing function, where the valid assembly of these building blocks forms a high-level structural parallel programming model. Therefore, instead of constructing a highly specialised but monolithic implementation of each skeleton/pattern that is optimised for a given target architecture, a building block based approach tries to construct high-level skeletons/patterns as a combination of a small set of simpler, efficient and re-usable building blocks.

Inspired by Backus' functional programming language [56], in [14, 15] a set of building blocks (RISC-pb^{2}l) have been proposed to support the modelling and implementation of high-

Table 2.1: An example set of well-known patterns [55]

Pattern Name	Definition
Task-Farm	Also known as master-slave, this represents the execution of a set of data items (the input tasks) processed by a same function f to generate a set of data items (the results).
Map	Represents split, execute and merge computation. A task is divided into sub-tasks, sub-tasks are executed in parallel according to a given skeleton, and finally the sub-tasks' results are merged to produce the original task's result.
Reduce	Combines every element in a collection into a single element using an associative combiner function.
Pipe	Represents staged computation in which different tasks can be computed simultaneously in different pipe stages. A pipe can have a variable number of stages, and each stage can possibly represent a nested pattern.
D&C	Represents divide and conquer parallelism. A task is recursively sub-divided until a condition is met, then the sub-task is executed and results are merged while the recursion is unwound.
For	Represents a fixed iteration where a task is executed a fixed number of times. In some implementations the executions may take place in parallel.
While	Represents conditional iteration, where a given skeleton is executed until a condition is met.
If	Represents conditional branching, where the execution choice between two skeleton patterns is decided by a condition.
Seq	Does not represent parallelism, but it is often a convenient tool for wrapping a block of code as a terminal node in a parallel pattern.

Table 2.2: Base building blocks for parallel instruction provided by [15]

Building Block Name	Syntax	Definition
Seq Wrapper	«code»	Wraps a sequential block of code into a RISC-pb^{2}l "function".
Par Wrapper	‹|code|›	Wraps any parallel block(s) of code into a RISC-pb^{2}l "function".
Parallel	$[*]_x$	Computes in parallel x identical programmes on x input items. No implicit synchronisation is applied among programmes at the end of computation.
MISD	$[*_1,...,*_x]$	Computes in parallel a set of x different programmes on x input items and produces x output items. No implicit synchronisation is applied among programmes at the end of computation.
Pipe	$\triangle_1 \cdot \ldots \cdot \triangle_x$	Uses x different programmes as stages to process the input items and to obtain output data items. Programme i receives input from programme $i-1$ and delivers results to programme $i+1$.
Reduce	$(g_{x\triangleright 1})$	Computes a single output item from x input items using a l ($l \geq 1$) level $k-$array tree. Each node in the tree computes (possibly commutative and associative) a $k-$array function g.
Spread	$(f_{1\triangleleft x})$	Computes x output items from an input by using a l ($l \geq 1$) level $k-$array tree. Each node in the tree uses the function f to compute k item out of the input data item.
1-to-N	$f_{1\triangleleft x}$	Sends data received on the input channel to one or more of the x output channels according to policy f where $f \in [unicast_p, broadcast, scatter]$ and $p \in [auto, roundrubin]$.
N-to-1	$g_{x\triangleright 1}$	Collects data from x input channels and delivers the collected data into a single output channel according to policy g where $g \in [gather, gatherall, reduce]$. *gatherall* waits for input from all input channels and delivers a vector of items, *de facto* implementing a barrier.
Feedback	$\overleftarrow{\triangle_{conds(z)}}$	Routes output data y relative to input data z ($y = \triangle(z)$) back to the input channel or drives them to the output channel according to the results of the evaluation of $conds(z)$. Also, it may be used to route back x output to x input channel.

level structured parallel programming frameworks. Following a RISC approach, the RISC-pb^{2}l set is architecturally independent. However, the implementation of the different blocks may be specialised to make the best use of the target architecture's peculiarities. Applying the RISC-pb^{2}l set, it is possible to model both general purpose parallel programming abstractions not usually listed in classical skeleton sets, and more specialised domain specific parallel patterns [15]. Table 2.2 represents the set of provided building blocks introduced in [15].

The difference between the algorithmic skeleton and parallel design pattern concepts is in the way the structured parallel programming abstraction is proposed. The former abstracts the parallel programming as parametric language constructs or library functions/objects. A skeleton based application can be constructed and customised through parameters by instantiating those "ready to use" library functions/objects. The latter introduces the abstraction through software engineering "recipes". Following such "recipes", it is possible to incorporate a given parallel pattern to construct a structured parallel application. From a programmer's point of view, the parallel design pattern approach requires more programming effort than the algorithmic skeleton approach.

Taking its origins from both algorithmic skeletons and parallel design patterns, the RISC-Pb^{2}l approach develops efficient, maintainable and portable parallel applications at a lower level in the software stack. Not only does it ease the application level programming by taking advantage of the skeleton/pattern approach, but it also eases the developing patterns to construct a structural parallel framework by generalising and distinguishing the semantic behaviour of each building block to construct specific skeletons/patterns.

Nonetheless, all approaches simplify the process of developing a parallel programming application by raising a certain level of abstraction from underlying hardware architectures.

Providing such an abstraction layer, these approaches distinguish "structured parallel programming" models, relieving the application programmer from the tasks associated with parallel pattern implementation and tuning. However, in the traditional, low level approaches such as MPI, OpenCL and OpenMP, the process of developing the parallel features within the application remains entirely the responsibility of the application programmer.

Moreover, such "structured parallel programming" models exploit parallel patterns as reusable components to implement complex applications by composing certified abstractions rather than designing ad hoc solutions. However, in the traditional, low level approaches a generated solution for implementing a parallel application is generally an ad-hoc solution and is not readily reusable when attempting to parallelise another application.

One of the objectives of such "structured parallel programming" models is to provide both functional and performance portability across different target architectures through abstraction from underlying architectures. However, generally, an ad-hoc solution provided by the traditional, low level approaches does not provide functional or performance portability when moving the application to different target architectures.

Examples of the Structured Parallel Programming Framework

Structural parallel programming models through *algorithmic skeletons* have long been considered a viable approach to introduce high-level abstraction to parallel programming. Algorithmic skeleton frameworks for CPUs are thoroughly analysed in [13]. As a recent algorithmic skeleton based framework, Intel Threading Building Blocks (TBB) [57] is defined as application-level libraries that target multi-core homogeneous architectures. TBB provides a high-level C++ template library for easy development of concurrent programmes. It exposes simple skeletons and parallel data structures to define computation tasks.

Realising the RISC-pb2l approach, FastFlow is an open source C++ based structural parallel programming framework built on top of POSIX threads. It has a set of pre-defined algorithmic skeletons that modell the main stream-based parallel patterns [14, 15, 58], namely:

- *Farm* skeleton applying in parallel the function modelled by an inner skeleton composition (the *farm workers*) to all the items appearing in an input stream and delivering the

Table 2.3: Correspondence between the RISC-pb^{2}l building block and FastFlow components

Building Block	Definition
«code»	`ff_node`.
‹|code|›	`ff_node` with encapsulated parallel code through `svc` function embedded in `ff_node` (e.g. offloading code to GPUs or using some OpenMP directives).
$[*]_x$	`ff_farm` with the same identical workers.
$[*_1, ..., *_x]$	`ff_farm` with different workers.
$\triangle_1$ . ... . $\triangle_x$	`ff_pipeline`
$(g_{x\triangleright 1})$	`ff_gatherer` encapsulating `ff_node` implementing g function.
$(f_{1\triangleleft x})$	`ff_loadbalancer` encapsulating `ff_node` implementing f function.
$f_{1\triangleleft x}$	`ff_loadbalancer` implementing the specified channel policy.
$g_{x\triangleright 1}$	`ff_gatherer` implementing the specified channel policy.
$\overleftarrow{\triangle_{conds(z)}}$	`wrap_around_method` for farms and pipelines.

results to its output stream.

- *Pipeline* skeleton applying in sequence the functions implemented by its inner skeleton compositions (the *pipeline stages*) to the items appearing in an input stream and delivering the results to its output stream.

When used at the topmost level in the skeleton composition, both *pipeline* and *farm* skeletons support a `wrap_around` method representing feedback from the output stream directly to the input stream.

FastFlow provides `ff_node` as an abstraction of parallel concurrent activity to process items delivered in an input stream and to convey the results to an output stream.

It provides `ff_loadbalancer` as a building block that schedules tasks from an input stream to a set of concurrent activities. Also, `ff_gatherer` has been provided as a building block that collects results from a set of concurrent activities in a single stream, either aggregated in a collection data structure or a sequence of values. These basic building blocks are used to implement the various high level skeletons provided as primitive classes to the application programmer. Table 2.3 demonstrates the realisation of RISC-pb^{2}l in the FastFlow framework [15].

With the emergence of GPGPU, recently, implementation techniques supporting the provision to the application programmer of *expandable* algorithmic skeleton sets have been demonstrated to implement algorithmic skeleton frameworks targeting heterogeneous multi-core architectures. Recent research on providing such skeleton based approaches includes SkePU [8], Thrust [9] and SkelCL [10].

The architecture of the above frameworks consists of two parts, namely, the front-end interface and the back-end implementation.

In the front-end interface each of these frameworks provide their own skeletons over the underlying heterogeneous (CPU/GPU) environment for the user. Each skeleton is generic in terms of both the type of data to be processed and the operations to be applied to the data. The operations that are applicable to the data can be either in the form of a predefined function provided by each approach or a user function, written by the programmer in the function template provided by each framework.

For the back-end, there are different implementations of each skeleton. Based on the information provided to the compiler command, an appropriate back-end will be selected for execution. Thrust supports the OpenMP directive for the multi-core CPU back-end and CUDA C language for the GPU back-end. SkePU supports OpenMP directives for the multi-core CPU back-end and CUDA C language and OpenCL language for the GPU back-end. SkelCL supports the OpenMP directive for the multi-core CPU back-end and OpenCL language for the GPU back-end.

The distribution policy of the input data in all of the above frameworks is as follows:

For the GPU back-end input, the data is divided based on the maximum number of available threads per block, L.

For the multi-core CPU back-end, the OpenMP *parallel for* directive is used which simply divides the input vector into blocks of $\frac{N}{K}$ where K is the maximum number of available threads and N is the size of the input data.

The underlying device will be selected by the user at compile time. It can either be a multi-core CPU or GPU but not both of them at the same time. Based on the selected back-end, the provided macro will expand into a CUDA kernel, C++ function or OpenCL kernel. Using the skeleton-based approaches, all frameworks provide an abstraction layer by hiding all the complexities of memory allocations, synchronisations, data distribution policies and execution plans from the user.

In Thrust there is an interoperability mode with CUDA C which allows the programmer to access it directly if needed. SKePU and SKelCL support lazy memory copying which transfers

the data from host to device and device to host upon request, which minimises the data transfer cost. Moreover, providing OpenCL back-end in SKePU and SKelCL supports the execution of a programme on any OpenCL-capable device, increasing the portability of applications.

2.3 Parallel Applications Optimisation

Assuming a reasonable level of abstraction is provided to generate an application, another issue would be:

- How to optimise the parallel execution of the application over available underlying hardware; and
- How much this process can be automated.

The former can be categorised as a scheduling problem and the latter can be categorised as auto-tuning the performance.

2.3.1 Scheduling System Over Heterogeneous Multi-core Architecture

Depending on the level of detail provided in a scheduling process, each technique will support one or more of the following parts:

- Load-balancer: Creates a balance by assigning an appropriate load of tasks to each node based on the throughput of that node.
- Processing element (PE) allocation: Is the process of deciding on which PE a new node should be executed.
- Node execution policy: Is the mechanism for choosing which node must be executed next in a given PE.

In the following we describe the scheduling approaches used in hybrid (CPU/GPU) architectures.

In [28] a predictive user-level scheduling approach has been introduced which dispatches the application with respect to the different speed-ups observed in the different processing units during previous executions, regardless of the data input. It uses a function-level granularity for scheduling, where the functions are provided by users for each CPU or GPU separately.

In another approach, a run-time system called Harmony [59] provides a scheduling phase which creates a dynamic mapping from kernels to heterogeneous architectures. In Harmony, the word kernel is analogous to functions or procedure calls in imperative programming languages. To address the variation of execution time across heterogeneous architectures, online

monitoring support is used to monitor the execution time of each kernel as well as the values of some of the input variables.

GPUSs [29] is an extension of the StarSs [60] programming model which allows the user to use multiple GPUs. To schedule a task, the main programme communicates with the corresponding worker thread by using event signalling. When a GPU becomes idle and a new task is ready for execution, the worker thread receives the necessary information to identify and invoke each task that is ready and to locate the necessary parameters for the execution of the task. StartSs and consequently GPUSs are `#pragma` annotation based languages similar to the OpenMP directive, which rely on a source-to-source compiler to generate offloadable tasks. However, each part of StarSs has its own run-time system and there are no integrated run-time systems for StarSs to allocate the tasks among both CPUs and GPUs.

In [30] a system called Anthill is used to provide a data-flow oriented framework in which applications are decomposed into a set of event-driven filters. For each event, the run-time system can use either GPUs or CPUs for its processing. Two modules are created on top of the filter framework: the device scheduler and the event executor. At run-time, the event executor creates threads and associates them with different devices. Each thread contacts the device scheduler, whenever idle, and determines an event to be processed within that thread based on the provided scheduling policies. Details of this algorithm can be found in [30].

The StarPU [22] provides a runtime system that is capable of scheduling tasks over heterogeneous, accelerator-based machines. Here a task refers to an independent unit of a programme applying certain computations to input data. A tag-based system is used to express the task dependency. The tasks submitted by an application are dispatched to one of the device drivers by the scheduler and then the driver offloads the proper implementation for the task. Different task scheduling policies are used here to assign the input data to each task. When a task is completed, other tasks that depend on it are released and an application call-back for the tasks is executed.

The StreamIt [61] is a language which provides high-level representations within the streaming domain by using Java syntax. It has a source-to-source compiler which generates a C code from the user-written code. The provided source-to-source compiler improves the performance of streaming applications via stream-specific analyses and optimisations. In [31] a software pipeline framework is introduced which efficiently maps StreamIt applications onto GPUs. It generates a scheduling formula which maps each instance of a *filter* (which is an independent unit of a programme) of the StreamIt onto exactly one SM of the GPU. This allows multiple threads to execute the *filter* which creates task parallelism. The optimised code for GPU is generated by modifying the StreamIt compiler to produce the CUDA code with the required drivers for each machine.

In [62] a dynamic OpenCL task scheduling algorithm has been proposed to schedule multiple kernels from multiple programmes on CPU/GPU heterogeneous platforms. The algorithm is more of a device oriented algorithm, as device utilisation is the main priority for allocating a device. In case of competition for a device, a kernel that is likely to best utilise the device will be allocated to the device. The decision to determine device utilisation for each kernel depends on kernel speed-up on that device which is predicted by a speed-up prediction algorithm. In this case, the efficiency of this approach depends on the accuracy of the predicting algorithm.

Where all the above approaches optimise an application performance by fine tuning the scheduling mechanism on heterogeneous applications, they do not consider:

- All the three parts of the scheduling process for an application; or
- Scheduling different components from multiple applications sharing the same resources.

Moreover, although it has a significant effect on application performance, scheduling is only one aspect of the coordination mechanism for parallel applications. In this case, using a certain level of abstraction to interact with a structural parallel framework, it might be possible to reuse the same technique not only for scheduling but also for different coordination aspects such as refactoring. This may lead to developping an autonomic optimisation of structured parallel applications.

2.3.2 Auto-Tuning Parallel Applications' Performance

As the subject of extensive research, several tools are currently available to auto-tune performance in parallel programming. Such research includes automatic parallelisation, performance visualisation, instrumentation and debugging as summarised in [63, 64]. One trend is to generate language-based tools to auto-tune parallel applications by using a static parsing mechanism to analyse the programme source code, determine the tunable parameters, such as the number of threads and identify parameter dependencies using pragma-based approaches [65, 66] or the generation of entirely new programmes [67, 68]. Typically, these approaches are limited to a single language or are highly specialised for a certain domain [69, 70].

An alternative problem is improving an application performance by optimising coordination. This may be approached with solutions based on runtime information regarding monitoring and reconfiguration.

In this thesis, we focus on the coordination problem for improving application performance related to the extra-functional properties of an application.

In [71] dynamic reconfiguration of grid-aware applications in the ASSIST programming environment has been proposed to capture and meet the quality of service requirements in Grid applications.

A framework for programming self-managing component-based distributed applications (self-*) is presented in [72]. It is organised as a network of management elements (MEs) that interact through events. Methods are categorised as sensors and actuators. The sensors determine environment changes through events generated by the management platform or by other application specific sensors. The actuators apply architectural changes, add, remove and reconfigure components and bindings between them via MEs.

The absence of the clean separation of coordination from computation in these approaches requires the direct involvement of programmers in the performance tools that should exist outside their programme. However, it is highly desirable for the optimisation of these coordination mechanisms, or extra-functional concerns, to be handled transparently from the user perspective [73]. Skeleton based frameworks provide an opportunity to achieve such a clean separation.

In [73, 74, 75] a behavioural optimisation technique for grid computing captures extra-functional concerns in an independent activity or the *autonomic manager*. These concerns are:

- Parallelism degree;
- Set-up and tuning;
- Dynamic load-balancing; and,
- Adaptation of parallelism patterns to different features of the target architecture.

The limitations of these parameters are specified by user applications in the form of a contract. Each component of the skeleton is equipped with its own autonomic manager that executes a control loop in one of two modes:

- *Passive mode:* Where a manager monitors the status of the current computation and awaits new contracts from its parents; and
- *Active mode:* Where contracts are received from the user application or a parent manager in the hierarchy and the appropriate autonomic actions are taken as required to maintain the contract.

Semi-formal models based on autonomic management are proposed for component based parallel and distributed programme development [76]. This programmer-oriented methodology is based on formal tools that permit reasoning about the programme and refinement.

Adaptive structured parallelism [77] (ASPARA) is a generic methodology to incorporate structural information at compilation into a parallel programme, which facilitates adaptation at runtime. The four phases are: *programming* during which API calls to ASPARA are parametrised, *compilation* when the structured parallel programme is instrumented, *calibration* which extrapolates node performance and selects the most appropriate node for the given

application, and *execution* which is responsible for selection and deployment on the chosen node.

Whilst these solutions have been developed in the context of CPU environments, they lack full support for multi-core CPUs and general-purpose computation on graphic processing units (GPGPU). In fact, higher-level approaches for the performance tuning of heterogeneous multi-core CPU/GPU applications have become an area of active research in computational science.

A queue monitoring heuristic is introduced in [78] to increase resource utilisation for divisible workloads performing numerical linear algebra on CPU and GPU resources that fit the pipeline parallel architectural pattern. Stochastic allocation of heterogeneous resources to pipeline components is proposed, which maximises throughput subject to queue stability. Following this work, a heterogeneous streaming pipeline implementation of these numerical linear algebra kernels over the FastFlow skeletal library [79] introduces adaptive throttling based on memory usage to coordinate the streaming pipeline and to dynamically allocate CPU and multi-GPU resources in a distributed memory cluster environment [80].

2.4 Research Gap

As stated in chapter 1, in [1, 2, 3, 4, 81] it has been stated that exploiting *heterogeneous* multi-core/many-core processor technologies for developing applications on different domains can provide up to 2 order of magnitudes speed-up. Therefore, using heterogeneous multi-core architectures can be considered as an essential requirement for future software developments.

The main challenge here is to find a programming model that provides a suitable level of abstraction, while still allowing a reasonable level of control over the execution of applications on the available heterogeneous hardware resources. Such a level of control is composed of a set of high-level decisions over a set of objectives for deploying parallel applications. These objectives can be varied, such as performance, cost, energy, fault-tolerance or even a combination of one or more of them. While some of these metrics are in line with each other, other combinations can be in conflict with each other.

Moreover, with continuous changes in the hardware technology layers, the priority of such objectives might change. For example, with the introduction of the cloud computing which provides affordable resources and software as services, the priority of an application cost and energy consumption can be more significant than a slight drop in performance. However, by introducing the heterogeneous multi-core architectures, an application can be capable of achieving a performance of up to two orders of magnitude. Such an improvement in performance can be significant enough to motivate developers to consider performance rather than cost or energy consumption to be a higher priority.

Such variations in the priority of objectives can have an affect on both the computation and

coordination units of applications. Moreover, regarding application performance concerns, it is already very difficult for classically-trained programmers to benefit from the performance offered by today's multi-core systems, and only highly-skilled programmers or those seeking the highest levels of performance are presently exposed to parallel programming techniques. In this case programmers will find it essentially impossible to exploit the mid-term/long-term developments that major hardware companies such as Intel and NVIDIA promise to deliver. The "dilemma" is that a large percentage of mission-critical enterprise applications will not "automagically" run faster on multi-core servers.

In this thesis we aim to answer the following question:

> With the continuous evolutions in heterogeneous multi-core/many-core architectures, how can we determine a mechanism with a suitable level of abstraction to hide the parallel programming complexity while being able to exploit the current architectures with the aim of being expandable on future hardware developments?

One hypothesis to achieve the research question would be to provide a high-level parallel programming model with a reasonable level of:

(i) Modularity to provide the separation of concerns;

(ii) Controllability to provide autonomy in coordination; and, if possible,

(iii) Extensibility to adapt to future changes in the underlying resources.

Such a high-level parallel programming model, would allow programmers to exploit the latest developments in heterogeneous multi-core/many-core architectures while still making it feasible to target the coordination over future hardware developments.

Realising such a high-level programming model can be considered as an empirical approach that targets the validation of the hypothesis.

As demonstrated in [14, 15], the modularity of the RISC-Pb^{2}l approach places it in a preponderant position to support the separation of concerns by dividing the functionality of each building block from the computation function that blocks must execute. Therefore, in this thesis instead of developing a new specific framework that supports abstraction and coordination over heterogeneous multi-core architectures, we provide SKIP as a generic strategy to target the autonomic coordination in building block based structural parallelism over heterogeneous multi-core architectures.

While the building block based structured programming aims to provide a certain level of abstraction to handle the synchronisation and communication mechanism in parallel programming, SKIP aims to exploit the building block approach to construct and coordinate structured parallel applications based on a set of provided extra-functional properties related to the parallel computation patterns in heterogeneous multi-core architectures.

Through the modularity provided by the building block approach, we aim to expand the building block approach to support the coordination of structured parallel applications in heterogeneous multi-core architectures. The fused SKIP methodology aims to automatically construct and instrument applications for dynamic coordination. We aim to provide a set of SKIP compliant coordination mechanisms for the behavioural structural parallelism framework to perform autonomic scheduling of components in heterogeneous devices and application performance related tuning. Through the interaction mechanism provided by the SKIP methodology, the behavioural framework aims to apply autonomic coordination optimisation through the information exchanges between the structured applications and the provided coordination engines. The applicability and efficiency of our methodology can be verified by a set of parallel applications implemented in a SKIP compliant building block framework.

Moreover, as RISC-Pb^{2}l approach creates independent blocks with determined functionalities for each block, by such modularity it abstracts the process of communication and synchronisation from the end user. However, the conceptual complexity of developing a parallel application is left unattended. In contrast with RISC-Pb^{2}l approach, the general purpose skeleton frameworks such as SkePU [8], Thrust [9] and SkelCL [10] not only hide the process of communication and synchronisation from end users, but also they hide the conceptual and coordination complexities of developing a parallel application via fixing the data structure format and computation functions. On the one hand, such fixation eliminates the extra designing and developing effort for thinking in parallel to generate a parallel application. On the other hand, it can limit the range of application domains that can be supported by such skeletons. As stated in [81], to generate an application by using such fixation, the desired performance improvement might not be achieved and it might complicate the process of developing an application due to the limitations in the data structure format and predefined computational functions.

Therefore, by using the RISC-Pb^{2}l approach to generate the autonomic behavioural framework, it would be possible to extend the range of applications that can be developed by structured parallel programming.

As stated in section 2.3.2, although different exisiting behavioural skeletons automate the coordination for heterogeneous parallelism, they mainly support non-functional properties, such as quality of service in grid computing systems.

> To the best of our knowledge, the autonomic management approach has not been applied on heterogeneous multi-core architectures for both extra-functional and non-functional properties.

Moreover, the existing autonomic management for parallel programming has mainly been applied over algorithmic skeleton approaches and they do not consider other structured programming models. In [14, 15] it has been demonstrated that the intrinsic characteristics of structured parallelism through the building block approach places this paradigm in a prepon-

derant position to support a reasonable level of abstraction by implicitly providing the communication and synchronisation for a parallel application. Also, it has been demonstrated that not only this approach supports common patterns in algorithmic skeletons, but it also supports other approaches such as the Google Map-Reduce model and domain specific parallel programming models. Therefore, PEI uses RISC-Pb^{2}l building block approach that is presented in [14, 15] as a structured parallel programming approach to support abstraction.

Our autonomic management system introduced via the SKIP-methodology supports coordination over heterogeneous multi-core architecture for both extra-functional and non-functional properties.

Such manager will arguably support autonomic coordination and optimise scheduling, load-balancing and structural configuration required in the by ParaPhrase project. Also, through SKIP methodology it would be possible to integrate other ParaPhrase coordination tools such as the refactoring one [82].

Chapter 3

SKIP Methodology for Coordinating Structural Parallel Programming

In this chapter we investigate a set of controlling parameters for modifying the coordination of a structured application affecting the application's performance and also resource utilisation in the system. Moreover, we extend the RISC-pb^{2}l approach by introducing a building block component that is capable of executing a computational function on accelerators in heterogeneous multi-core architectures. Combining both, we propose a SKIP (Structural Composition and Interaction Protocol) as a systematic methodology that is capable of automatically constructing and modifying the controlling parameters. Furthermore, we explain the procedure of autonomic coordination of a structured parallel application incorporated with the SKIP methodology. Finally, we end the chapter by summarising the proposed SKIP methodology.

3.1 Controlling Parameters

Separation of concerns and controllability are the key factors for tackling the research gap.

Structured programming is a viable and effective means of providing the separation of concerns, as it subdivides a system into building blocks (modules, skids or component) that can be independently created, and then used in different systems to drive multiple functionalities.

Moreover, by subdividing a structured application into a set of building blocks, it is possible to separate the coordination, communication and synchronisation of an application from the computational functionalities it provides. In this case, with no deviation from an application computation, it is viable to tune the application coordination both statically and dynamically. By defining tunable controlling parameters with a valid configuration range for each parameter, we can tune the coordination features with no changes in computation functions.

In this thesis, we investigate a set of high-level frameworks including FastFlow, SKePU, Thrust and Intel TBB that support abstraction mechanisms. We extract the controlling pa-

rameters where the variation can affect the application performance, along with the resource utilisations in the system. Table 3.1 represents a set of parameters for modifying the coordination of a structured application. The chosen parameters in this table will affect the coordination with no changes in the application computation functions. However, general purpose skeleton frameworks such as SKePU and Thrust use two other parameters, namely:

- application input data structure (Array(1D, 2D, 3D),...).
- application input data type (Integer, String,...).

to tune the performance and resource utilisation by enhancing data offloading process on GPU devices As stated in [81], using these parameters requires some modifications in application computation functions. Therefore, we did not include them in the controlling parameters table. As stated in Section 2.2 in Chapter 2, in addition to syntax, each RISC-pb^{2}l building block component has *semantic behaviour* which determine the expected service that the component delivers (component definition column). While the computation function of a building block can be varied for different applications, the semantic behaviour of the building blocks remains intact. In this case, by understanding the behaviour of the building block components and their valid compositions, the process of constructing a structured application and coordinating it regarding these controlling parameters can be automated. As stated in Table 3.1 such parameters can be determined as environmental conditions, on demand user conditions, resource conditions and application conditions. These conditions can be determined:

- Statically before application execution to impose specific preparation; or
- Dynamically over time via feedback from applications and resources to determine changes in a resource or an application status.

Information about how well an application is performing has to be monitored during the application execution and compared somehow with the input that represents information about what the system should do. Such a comparison provides the required information to produce an appropriate controlling signal to be able to adapt to the variations in coordination parameters.

3.2 Extending RISC-pb^{2}l Over Heterogeneous Architectures

To mitigate the research gap, we have applied the RISC-pb^{2}l building block approach and orchestrated it with our provided coordination mechanisms. In order to efficiently perform coordination in heterogeneous (CPU/GPU) environments, we have introduced a new building block called *HWrapper*.

Table 3.1: A brief example of possible scenarios for *Control-required Conditions*.

Control-required Conditions	Scenario
Resource Conditions	Resource Status (Idle or Busy), Resource Availability, Resource Capacity, Resource Constraints, Resource Type (CPU or GPU).
Application Conditions	Application Status (suspend or Resume), Application Priority, Application Component State (busy,idle), Application Component Status (ON or OFF).
Environment Conditions	Framework Choice (OMP, MPI,CUDA, OPENCL, or any combinations of these), Platform Choice (CPU,GPU, Hybrid).
User Conditions	Observation Cycle, Application Execution Order Policy, Application Priority Policy.

Hwrapper: Encapsulates an accelerator code in a RISC-pb^{2}l "function". It differentiates between embedded parallel blocks offloading on CPUs only with those offloading on accelerator devices.

The *Hwrapper* is a subset of *ParWrapper* introduced in RISC-pb^{2}l.

ParWrapper will execute the block with no separation of the execution environment type. Therefore, the coordination for allocating resources, offloading and coordinating on a device must be provided within the computation functions encapsulated by *ParWrapper*. The intermixture of embedded function types in a *ParWrapper* block prevent an application from dynamic coordination in a heterogeneous environment. In this case, no information is provided for a structured application about the encapsulated function it is executing in that specific block.

Hwrapper has the capability to distinguish between the accelerator based components and CPU-only components. The separation of blocks by type in a heterogeneous (CPU/GPU) multi-core architecture provides the extra information required to further optimise and tune the coordination of a structured application.

Borrowing the concept of higher order functions/combinators distinction proposed for sequential building blocks in Backus' functional programming (FP) [56], the RISC-pb^{2}l approach aims to provide a similar "algebra of programmes" that is suitable for supporting parallel programme refactoring and optimisation [14, 15].

Towards this aim, we propose a new grammar to formalise the valid compositions of RISC-pb^{2}l building blocks executed in heterogeneous environments. Listing 3.1 represents the valid

composition rules for the grammar to generate a structured application supported by RISC-pb^{2}l over heterogeneous architectures.

In Appendix A we have demonstrated the validation of our grammar in detecting patterns supportable by RISC-pb^{2}l.

$$
\begin{array}{lcl}
\square & ::= & \blacktriangle \mid \blacklozenge \\
\blacktriangle & ::= & \triangle \mid \triangledown \\
\triangledown & ::= & \triangle . \triangledown \mid \triangle . \triangle \\
\blacklozenge & ::= & \blacktriangle . \blacklozenge \mid \diamond \\
\triangle & ::= & \dashv^{1n} . \vdash^{n1} \mid \dashv^{1n} . \pentagon^{n} . \vdash^{n1} \mid \overleftarrow{(\dashv^{1n} . \vdash^{n1})_c} \mid \overleftarrow{(\dashv^{1n} . \pentagon^{n} . \vdash^{n1})_c} \mid \overleftarrow{(\triangledown)_c} \\
 & & \mid \text{«code»} \mid \text{‹ | code | ›} \mid \text{« ‖ code ‖ »} \\
\diamond & ::= & \dashv^{1n} . \bigcirc^{n} \mid \overleftarrow{(\models_1^{n1} . \blacktriangle . \dashv^{1n} . \pentagon^{n})_c} \mid \overleftarrow{(\models_1^{n1} . \dashv^{1n} . \pentagon^{n})_c} \\
\bigcirc^{x} & ::= & \pentagon^{x} \mid \pentagon^{x} . (\vdash^{m1})^{k} . \bigcirc^{k} \mid \pentagon^{x} . (\dashv^{1z})^{x} . (\vdash^{x1})^{z} . \bigcirc^{z} \\
 & & \mid \pentagon^{x} . (\dashv^{1z} . \pentagon^{z})^{x} . (\vdash^{x1})^{z} . \bigcirc^{z} \mid (\square)^{x} \\
\pentagon^{x} & ::= & (\blacktriangle)^{x} \mid (\blacktriangle)^{x} . \pentagon^{x} \\
\dashv^{1x} & ::= & \dashv_{mk}^{1x} \mid f_{1 \triangleleft x} \mid (f_{1 \triangleleft x}) \\
\dashv_{mk}^{1x} & ::= & (\dashv^{1m})^{k} \mid (\dashv^{1m} . \pentagon^{m})^{k} \\
\vdash^{x1} & ::= & \vdash_{mk}^{x1} \mid g_{x \triangleright 1} \mid (g_{x \triangleright 1}) \\
\vdash_{mk}^{x1} & ::= & (\vdash^{m1})^{k} . \vdash^{k1} \mid (\vdash^{m1})^{k} . \pentagon^{k} . \vdash^{k1} \\
(*)^{x} & ::= & [*]_x \mid [*_1, ..., *_x]
\end{array}
$$

Listing 3.1: A grammar for automatically generating RISC-pb^{2}l based structured programming. Here, $x = m \times k$ and $\models_1^{n1}$ is a $\vdash^{n1}$ with an extra channel to capture the data from the input channel.

A $\square$ represents a RISC-pb^{2}l application. We represent *Parwrapper* as ‹|code|›, *Hwrapper* «‖code‖» and *Seqwrapper* as «code» in order to distinguish between them.

Depending on the arity of an input/output channel, the valid compositions are mainly categorised into two classes: *reduction composition* (▲) and *non-reduction composition* (◆).

In a reduction composition (▲) both input and output channel arity is one, while in a non-reduction composition (◆) the input channel arity is one but the output channel arity is greater than 1. We have separated reduction composition from non-reduction composition as they have different usage. Any set of reduction compositions can be assembled with each other through a one-to-one channel (.) to generate a pipe construction. However, non-reduction composition can only be used as the last stage of a pipe construction. Also, the feedback system for a non-reduction composition requires an extra $N - to - 1$ combinator at the beginning of an application to direct the output data to the application input channel. As for all reduction compositions the input and output channel arity are one and nested parallel patterns inside a reduction composition requires a $1 - to - N$ combinator to distribute the tasks among the parallel computation units. It also requires a $N - to - 1$ combinator to collect the task from the parallel computation units. These two combinators are applied to satisfy the reduction composition rule. The number of computation units running in parallel, either in the form of

MISD or *parallel* should correspond to the input channel arity providing the tasks for those units and also, when applied, to the output arity channel receiving the tasks from those units.

The parallel augmentation of reduction compositions can be handled by either the combination of $N-to-1$ and $1-to-N$ combinator ($\dashv^{1n}\,.\,\vdash^{n1}$) or an additional nested parallel computation ($\dashv^{1n}\,.\pentagon^{n}.\,\vdash^{n1}$) when required. Unlike the reduction composition for a non-reduction composition, there is no need for an $N-to-1$ combinator. This will extend the range of structural pattern based applications that can be supported by the grammar presented in Listing 3.1.

The $(*)^x$ applies the parallel computation in the form of MISD or parallel to augment the level of parallelism. It can be applied for reduction composition parallelism ($(\blacktriangle)^x$), non-reduction composition ($(\blacklozenge)^x$) or combinators ($(\vdash^{m1})^k$ and $(\dashv^{1m})^k$). Augmenting the level of parallelism for combinators contributes to the arity of the combinators' channel. In this case, while the input arity channel of a $1-to-x/x-to-1$ combinator shrinks from x to m by a factor of k, at the same time, k instances of a $1-to-m/m-to-1$ combinator is provided to reconstruct the $1-to-x/x-to-1$ combinator. When possible, such a shrinking technique for a combinator helps to augment the level of parallelism in an application. For example, the arity of the input channel for the parallelised combinator $(\dashv^{1m})^k$ is equal to $k \times m$. Therefore, a $1-to-N$ combinator can break into a k set of $1-m$ combinators $(\dashv^{1m})^k$ to further augment the level of parallelism for distributing data for parallel computation. Also, each $1-m$ combinator can be equipped with a set of parallel computations to support any prerequisite parallel computations over dispatched data ($(\dashv^{1m}\,.\pentagon^{m})^k$). Also, using this technique in the non-reduction composition, it is possible to shrink the parallel level by applying $\pentagon^x\,.\,(\vdash^{m1})^k\,.\,\bigcirc^k$ where the parallel level is reduced by a factor of m from x to k.

Although the number of parallel computation units in a non-reduction composition should correspond to the number of input arity channels of its provided $1-to-N$ combinator, having no $N-to-1$ reduction combinator provides flexibility in non-reduction composition to support a transient composition. A transient composition can be represented as a composition of $(\dashv^{1z})^x\,.\,(\vdash^{x1})^z$, changing the arity of the input channel for the next computation unit. Using transient composition it is possible to change the parallel augmentation level for the next computation unit. In this case, as stated in the composition rule $\pentagon^x\,.\,(\dashv^{1z})^x\,.\,(\vdash^{x1})^z\,.\,(\bigcirc^z)$, the parallel augmentation level will change from x for $\pentagon^x$ to z for $\bigcirc^z$. It is also possible to apply a further computation on transient composition by replacing $(\dashv^{1z})^x$ with $(\dashv^{1z}\,.\pentagon^{z})^x$ in the transient composition $(\dashv^{1z})^x\,.\,(\vdash^{x1})^z$. The transient composition is useful for detecting patterns like *Google Map-reduce*, allowing an arbitrary level of parallel augmentation for the map and reduce stages.

The concept of $(\vdash^{y1})^z$ is equivalent to $(\dashv^{1z})$, meaning that z number of $\vdash^{y1}$ is provided to direct the output channel of the y computation unit into 1 channel. Therefore, these provided z output channels can be used as input channels for either another parallel computation unit, or a

$\vdash^{z1}$ for further reduction.

Moreover, as the reduction composition has a deterministic one-to-one input/output arity channel, it is possible to merge to parallel computation with the same level of augmentation ($\pentagon^n \,.\, \pentagon^n$). This will optimise the parallelism by removing unnecessary reduction and spread combinators when the level of augmentation for both parallel computations are the same. Also, $(\vdash^{k1})^m . \vdash^{m1}$ $(\vdash^{k1})^m \,.\, \pentagon^m . \vdash^{m1}$ can be applied to support the parallel reduction which optimises the reduction procedure by augmenting the level of parallelisation.

Feedback construction is either possible for an entire pattern or a subset of pipe composition. Feedback can also be applied for single reduction composition, however, the feedback for a single terminal construction including *secwrapper*, *parwrapper* and *hwrapper* is not applicable as feedback, and in these cases can be represented in the form of recursion on their computational function. When a non-reduction pattern is applied, the reduction combinator $\models_1^{n1}$ is required to first collect and redirect the computed data to the specified feedback compositions. The building block $\models_1^{n1}$ represents $\vdash^{n1}$ with an extra input channel to capture the input data not redirected by feedback. Therefore, $\models_1^{n1}$ can be applied at the beginning of a non-reduction composition to support the feedback composition for it.

3.3 Structural Composition and Interaction Protocol (SKIP)

We have designed and fully implemented a *new* structural composition and interaction protocol called SKIP, as a text-based protocol (with content representation in a human-readable format) to control extra-functional and non-functional features in behavioural skeletons.

Inspired by Javascript Object Notation (JSON) [83], SKIP notations are a set of *KEY:VALUE* pairs required to *i)* define the architecture of a structured parallel programme; *ii)* detect the extra functional features; and *iii)* adjust the predefined controlling conditions.

Each building block in RISC-pb^{2}l, regardless of the computational function it delivers, has a predetermined semantic behaviour. Therefore, through SKIP notations, it is possible to provide a descriptive representation of a building block regarding the predefined semantic behaviour of that building block.

SKIP Compliant Building Block Format A SKIP compliant building block is composed of a set of `KEY:VALUE` pairs that describe the semantic behaviour of the block. The combination of `KEY:VALUE` pairs for describing a building block depends on the semantic behaviour of the building block. The information that is determined by a `KEY:VALUE` pair can be classified as follows.

Structural Meta-data: Determines the required information for capturing the structural features of each building block in a structured programme. Table 3.2 explains the `KEY:VALUE`

Table 3.2: Structural meta-data

KEY	VALUE
APPLICATION_NAME	Is a string value determining the name of the application.
COMPONENT_NAME	Is a string value determining the name of a specific building block.
STRUCTURE_TYPE	Is a string value determining the type of specific building block according to the provided types in RISC-pb^{2}l. It can be one of the types provided in Table 2.2.
NODE_ADDRESS	Is a string value determining a URL address of a node in the directed graph of a structured application.
FUNCTION_NAME	Is a string value representing the name of the computational functions executed by the building block.
CH_POLICY	Is a string value representing one of the policies provided for combinators. It is only applicable to combinators.
CH_MULTIPLICITY	Is an integer value (>= 1) representing the multiplicity of a combinator. It is only applicable to combinators.
FEEDBACK	Is a boolean value that represents whether or not the component should reroute the output data relative to input data.
PAR_LEVEL	Is an integer value that determines the number of identical components to be executed in parallel.
SETUP_FUNCTION_NAME	Is a string value representing the set up function name required to prepare the device building block in the host site.
KERNEL_PATH	Is a string value representing the URL in the location of the kernel file executed on the device.

pairs for structural meta-data in detail.

Control Parameters: Is a set of tunable parameters that represent the coordination modification options for each building block. Table 3.3 explains the `KEY:VALUE` pairs for control parameters in detail.

Performance Metrics: Determines the extra functional requirements for each building block. This information is extracted during an application execution. Table 3.4 explains the `KEY:VALUE` pairs for performance metrics in detail.

SKIP Compliant Constraints SKIP not only determines a descriptive representation of the building block semantic, but also adds the concept of constraint to RISC-pb^{2}l programming. A constraint object determines a certain configuration or policy of a structured application or a specific device. If an application or device is not determined, the provided policy applies to all available devices or applications. Table 3.5 demonstrates the detailed explanation of each

Table 3.3: Control parameters

KEY	VALUE
WORKLOAD	Is a list of floating point values that determine a workload distribution for each n-output channel. It is applicable to $1toN$ combinators.
APPLICATION_STATUS	Is a string value that determines the status of a structured application. It can be either SUSPEND, meaning that an application is suspended or RESUME, meaning that an application is running.
PRIORITY	Setting per application, PRIORITY represents an integer value that determines the application priority in the system. The smaller the number the higher the application priority.
CH_BOUND	Is an integer value that determines the maximum number of tasks a building block can buffer in an input/output channel. The default value of CHANNEL_BOUND for each building block is unlimited.
PROCESSOR_NUMBER	Is an integer number that determines the processor number for a building block.
DEVICE_NUMBER	Is an integer number that represents the device number for a device building block.
DEVICE_TYPE	Is a string value that determines the device type for a device building block.
COMPONENT_STATUS	Determines the status of a component in the system. It can be either *ON* or *OFF*. The former means that the component is active, while the latter represents that the component has been suspended.
MASKING	Is a list of boolean representing which building block is eligible to receive a task from a $1-to-N$ combinator. It is only applicable for $1-to-N$ combinators.
PROFILE_AND_TUNE	Is a list of boolean representing whether or not the coordination mechanism should be applied.

Table 3.4: Performance metrics.

KEY	VALUE
PROCESSED_TASKS	Is a long integer value that determines the total number of tasks processed so far by this component.
Queue_INPUT	Is a long integer value that determines the number of tasks inside the input channel yet to be processed.
Queue_OUTPUT	Is a long integer value that determines the number of tasks inside the output channel processed by a component.
Queue_LIMIT	Is a list of integer values that represent the maximum task limitation that can be buffered to be processed by a component.
COMPONENT_LAST_PROCESSING_TIME	Is a floating point value that represents the service time of the last execution.
COMPONENT_DISTRIBUTION_TIME	Is a floating point value that represents a list of service time frequency distributions during the building block execution.
TOTAL_COMPONENT_ACTIVE_TIME	Is a floating point value that represents the total service time of a component on which the component was executing a task.
ELAPSED_TIME	Is a floating point value that represents the total execution time of a component since the beginning of the application execution.
ASSIGNED_DEVICE_NUMBER	Is an integer number that represents the device number that is assigned to a building block.
END_RECEIVED	Is a boolean value that represents whether or not a component has exhausted all its allocated tasks.
CH_IN	Is an integer number that represents the number of tasks inserted to a 1*toN* combinator. It is only applicable for 1*toN* combinators.
CH_OUT	Is an integer number that represent the number of tasks sent by a 1*toN* combinator. It is only applicable for 1*toN* combinators.
POP_DELAY_TIME	Is a floating point value that represents the total waiting time of a building block for an empty input channel.
PUSH_DELAY_TIME	Is a floating point value that represents the total waiting time of a building block for a full output channel.
POP_DELAY_COUNT	Is a long integer value that represents the total number of times a building block waits for an empty input channel.
PUSH_DELAY_COUNT	Is a floating point value representing the total number of times a building block waits for a full output channel.

Table 3.5: Constraint configurations

KEY	VALUE
PRIORITY_POLICY	Is a string value that provides the type of priority used for an application. It can be either a FIXD, meaning that the application priority will not change over time; or VARIABLE, meaning that the application will age over time based on the provided ageing algorithm.
MAXIMUM_BB_PER_DEVICE	Is an integer value that provides the maximum number of building blocks that can be allocated on a device without any drop in performance or causing the the system to crash.
DAMPING_RATIO:	Is a floating point value that determines the level of sensitivity. The higher the value, the less sensitive the system. However, selecting a number that is too small will overshoot the desired output.
SAMPLING_MODE	Is a string value that controlles the level of information monitoring for service time, push delay and pop delay frequency distribution. It can be either AGGRESSIVE where comprehensive information is gathered on every individual task completed by a component; or SPARSE where statistical sampling is performed at a specified sampling rate for only a certain fraction of the completed tasks. This is intended to be the default mode.

KEY:VALUE pair for constraint configuration.

SKIP Compliant RISC-pb^{2}l Structured Programming As indicated in Listing 3.2 we have infused the SKIP methodology with the building block grammar presented in Listing 3.1, to introduce a descriptive representation of SKIP compliant RISC-pb^{2}l structured programming where each building block is defined by a set of SKIP compliant KEY:VALUE pairs.

```
 SKIP  ::= '{' □  ',' APPLICATION_NAME:STRING ',' PRIORITY:INT ','
      PROFILE_AND_TUNE: ('true' | 'false') ',' SAMPLING_MODE:STRING | CON '}'
□   ::= ▲ | ◆
▲   ::= △ | ▼
▼   ::= 'ReductionPipeComposition'':' '{' SEQ ',' ▽ '}'
▽   ::= △.▽ | △.△
⬡   ::=  'ReductionComposition' ':' '{' SEQ ',' ⫞^1n ',' ⊢^n1 '}'
      | 'ReductionComposition' ':' '{' SEQ ',' ⫞^1n ',' ⬠^n ',' ⊢^n1 '}'
△   ::=  ⬡ | 'Feedback' ':' '{' (▼ | ⬡) '}'
      | 'Sequential' ':'  '{' SEQ '}'
      | 'Hsequential' ':' '{' HSEQ '}'
◆   ::= 'NonReductionPipeComposition' ':' '{' SEQ ',' ◈ '}' | ◇
◈   ::= ▽.◈ | △.◇
◇   ::= 'NonReductionComposition' ':' '{' SEQ ',' ⫞^1n ',' ○^n '}'
      | 'Feedback' ':' '{' 'Composition' ':' '{' SEQ ','
    ( (⊨^n1_1 ',' (△ | ▽) ',' ⫞^1n ',' ⬠^n) | (⊨^n1_1 ',' ⫞^1n ',' ⬠^n) ) '}' '}'
```

$\bigcirc^x$ ::= $\pentagon^x$ | $(\square)^x$ | 'TransientComposition' ':' '{' SEQ ',' $\pentagon^x$ ','
(
$(\vdash^{m1})^k$ ',' $\bigcirc^k$
| $(\dashv^{1z})^x$ ',' $(\vdash^{x1})^z$ ',' $\bigcirc^z$
| ('Composition' ':' '{' SEQ ',' $\dashv^{1z}$ ',' $\pentagon^z$ '}')^x ',' $(\vdash^{x1})^z$ ',' $\bigcirc^z$
)
'}'

$\dashv^{1x}$ ::= $\dashv^{1x}_{mk}$ | '1ToNCom' ':' '{' SSEQ '}'
| '1ToNCom' ':' '{' SSEQ ',' 'Filter' ':' '{' SEQ '}' '}'

$\dashv^{1x}_{mk}$ ::= $(\dashv^{1m})^k$ | ('SpreadComposition' ':' '{' SEQ ',' $\dashv^{1m}$ ',' $\pentagon^m$ '}' $)^k$

$\vdash^{x1}$ ::= $\vdash^{x1}_{mk}$ | 'NTo1Com' ':' '{' RSEQ '}'
| 'Nto1Com' ':' '{' RSEQ ',' 'Filter' ':' '{' SEQ '}' '}'

$\vdash^{x1}_{mk}$ ::= 'ReduceComposition' ':' '{' SEQ ',' $(\vdash^{m1})^k$ ',' $\vdash^{k1}$
| $(\vdash^{m1})^k$ ',' $\pentagon^k$ ',' $\vdash^{k1}$ '}'

$\pentagon^x$ ::= $(\blacktriangle)^x$ | $(\blacktriangle)^x$ ',' $\pentagon^x$

$(*)^x$ ::= 'Parallel' ':' '[' '{' 'parLevel' ':' 'x' ',' $*$ '}' ']'
| 'Parallel' ':' '[' $\underbrace{\text{'\{' } *, \ldots, * \text{ '\}'}}_{\text{x times same } *}$ ']'
| 'MISD' ':' '[' '{' $*_1$ ',' ... ',' $*_x$ '}' ']'

```
SEQ ::= (KEY ':' VALUE )  |  SEQ ',' SEQ
HSEQ ::= (HKEY ':' VALUE ) |  HSEQ ',' HSEQ
SSEQ ::= (SKEY ':' VALUE ) |  SSEQ ',' SSEQ
RSEQ ::= (RKEY ':' VALUE ) |  RSEQ ',' RSEQ
CON ::= ('Device' | 'Application') ':' '{'CNT'}'
     | (CONST ':' VALUE ) | CON ',' CON
CNT ::= ('NAME' ':' VALUE ), (CONST ':' VALUE )
     | ('NAME' ':' VALUE ), (CONST ':' VALUE ) , CNT
KEY ::= 'APPLICATION_NAME' | COMPONENT_NAME |'COMPONENT_TYPE'
     | 'NODE_ADDRESS' | 'FUNCTION_NAME'
     |'PROCESSOR_NUMBER' | 'COMPONENT_STATUS' | 'CH_BOUND'
     | 'APPLICATION_STATUS' | 'PROCESSED_TASKS' | QUEUE_LIMIT
     | 'PUSH_DELAY_TIME' | 'COMPONENT_LAST_PROCESSING_TIME'
     |'POP_DELAY_TIME' | 'TOTAL_COMPONENT_PROCESSING_TIME'
     | 'COMPONENT_TIME_DISTRIBUTION' | 'ASSIGNED_DEVICE_NUMBER'
     | 'END_RECEIVED' | ELAPSED_TIME | 'PUSH_DELAY_COUNT'
     | 'POP_DELAY_COUNT' |'QUEUE_INPUT' | 'QUEUE_OUTPUT'
HKEY ::= KEY | 'SETUP_FUNCTION_NAME' | 'KERNEL_PATH'
     | 'DEVICE_NUMBER' | 'DEVICE_TYPE'
RKEY ::= KEY |'CH_POLICY' | 'CH_MULTIPLICITY'
SKEY :=  RKEY | 'WORKLOAD' | 'CH_IN' | 'CH_OUT'
CONST ::= 'PRIORITY_POLICY' | 'MAXIMUM_BB_PER_DEVICE'
     | 'DAMPING_RATIO'| 'SAMPLING_MODE'
VALUE ::= ATOMIC | '[' VALS ']'
VALS  ::= ATOMIC | ATOMIC ',' VALS
ATOMIC ::= STRING | NUMBER | 'true' | 'false' | 'null'
STRING ::= ' " ' CHARS ' " '
```

```
CHARS ::= CHAR | CHAR CHARS
CHAR ::= 'any Unicode character except " or \ or control character'
     | '\"' | '\\'| '\/' | '\b' | '\f' | '\n' | '\r' | '\t'
     | '\u four hexadecimal digits'
NUMBER ::= INT | INT FRAC | INT EXP | INT FRAC EXP
INT ::= DIGIT | D1-9 DIGITS | '-' DIGIT | '-' D1-9 DIGITS
FRAC ::='.' DIGITS
EXP ::= EVAL DIGITS
DIGITS ::= DIGIT | DIGIT DIGITS
DIGIT ::= '0' | D1-9
D1-9 ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Listing 3.2: SKIP definition for building block grammar provided in Listing 3.1. Here, $x = m \times k$ and $\models_1^{n1}$ is a $\vdash^{n1}$ with an extra channel to capture the data from the input

Taking advantage of structured pattern based parallelism through the RISC-pb^{2}l approach, SKIP represents the structural demonstration of an application. In this case, while control or data flow demonstrations of an application with different component instantiations can be represented in a directed cyclic/acyclic graph, the structure of the application generated by a SKIP object is represented in a tree format.

Each SKIP compliant object is represented in a tree structure format where each non-terminal nodes represents one of the proposed compositions in Listing 3.2 and the terminal nodes (leaves) are one of the terminal building blocks. The terminal building blocks are `Sequential, HSequential, Spread, Reduce, g-pol` and `d-pol`. The intermediate nodes can represent a nested composition.

As SKIP notations are specific for different types of terminal node, in Listing 3.2, the `Sequential, HSequential, Spread(or d-pol)` and `Reduce(or g-pol)` are assigned to the `SEQ, HSEQ, SSEQ` and `RSEQ` parameters with different key sets `KEY, HKEY, SKEY` and `RKEY` respectively to distinguish the valid SKIP notations for each node type. Moreover, `CON` is used to build a constraint object that represents a set of `KEY:VALUE`, where the valid constraint keys are represented by a `CONST` parameter. Also, the `CNT` parameter is applied to define a constraint object for a specific application or device. Similar to JSON, the definition of `VALUE` and `ATOMIC` parameters including `STRING, NUMBER, true, false` and `null` are very much like C or Java, except that the octal and hexadecimal formats are not used.

Parwrapper is not considered as a separate terminal building block. The coordination mechanism for the *Parwrapper* is the same as *Seqwrapper*, as there is no awareness about the type of computation function it encapsulates. Therefore, in a SKIP compliant object, a *Parwrapper* component is demonstrated by a `Sequential` building block for which the computation function has encapsulated any nested parallelism.

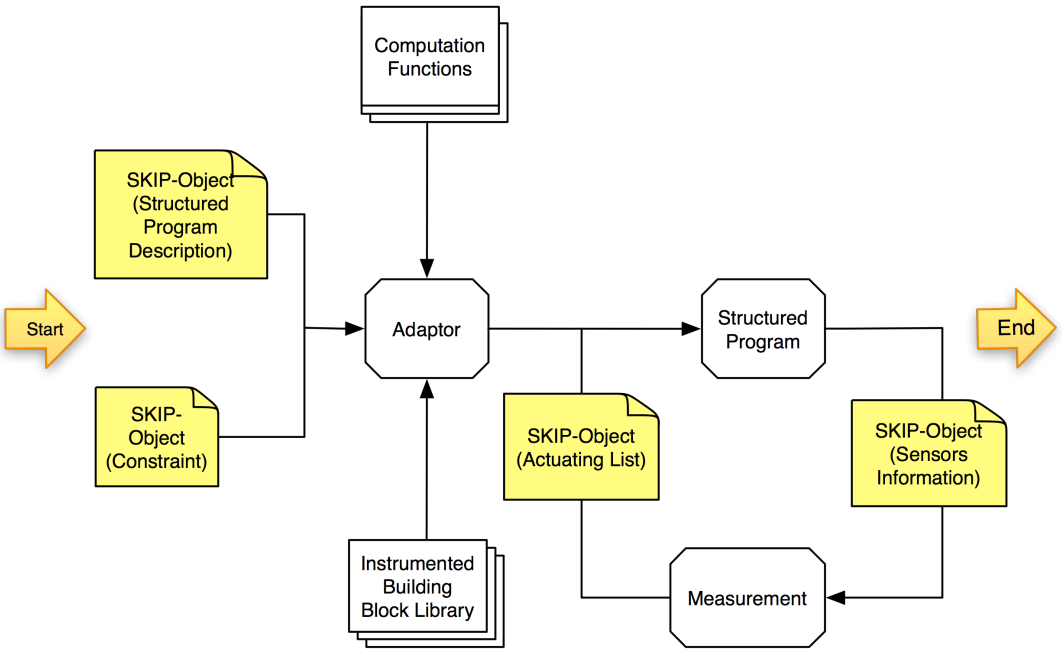


Figure 3.1: Schematic view of construction and execution of a structured program using SKIP compliant autonomic behavioural system

3.4 SKIP Compliant Autonomic Behavioural Framework

We design an autonomic behavioural framework that incorporates the SKIP methodology with RISC-pb^{2}l based programming model to coordinate structural applications by applying a set of predefined coordination mechanisms to optimise the coordination conditions provided in Table 3.1.

Figure 3.1 represents the construction and execution cycles of a structured programme developed through a SKIP compliant autonomic behavioural framework. The descriptive construction and autonomic coordination require interaction with the RISC-Pb^{2}l library. We design such interaction by fusing the SKIP methodology inthe RISC-Pb^{2}l library as stated in listing 3.2.

An *Adaptor* is an interface method which can generate a structured application by having:

- A SKIP compliant object representing the application description according to the descriptive representation of the RISC-Pb^{2}l grammar provided in Listing 3.2. The programme description contains structural meta-data features represented in table 3.2 to demonstrate the structural composition of an application. Also, it can contain controlling parameters represented in table 3.3 to statically set coordination features for an application. Appendix B represents examples of SKIP-compliant objects for applications that have been introduced in Chapter 5.

- A SKIP compliant object representing the environmental configuration in the form of

constraints. This is an optional file containing features represented in table 3.5. Appendix C, section C.3 represents examples of SKIP-compliant objects for constraint configurations for applications that are presented in Chapter 5.

- The instrumented RISC-pb2l library presented in Listing 3.2.
- A set of computation functions executed by building blocks in RISC-Pb^{2}l library. Generated by end-users, these functions represent the actual computations that should be applied by different building blocks on input data. The computation function signature depends on the implementation of RISC-Pb^{2}l library.

Once an *Adaptor* constructs a structured application, the '*Measurement* unit monitors the application and tunes the controlling parameters automatically and periodically during the application life cycle. The *measurement* unit extracts performance metrics periodically, analyses the extracted performance metrics, proposes new configurations for controlling parameters and then imposes these configurations on the constructed structured applications.

The *Measurement* unit contains a set of coordination engines for analysing and proposing new configurations for controlling parameters based on the extracted performance metrics. The extracted performance metrics and the proposed configurations are in the form of SKIP-compliant object files and are called *sensor information* and *actuating list*, respectively. These files are generated according to the e building block grammar presented in Listing 3.2.

For each building block, an *actuating list* file contains a structural meta-data information presented in table 3.2 and controlling parameters presented in table 3.3 according to the rules presented in Listing 3.2. The controlling parameters represent the new configuration proposed to tune the coordination. The structural meta-data information is used to identify component location in the structured application that the new configuration must be imposed on. Appendix C, sectionC.2 represents examples of SKIP-compliant sensor files for application descriptions that have been introduced in Chapter 5.

For each building block, a *sensors information* file contains a structural meta-data information presented in table 3.2 and performance metrics presented in table 3.3 according to the rules presented in Listing 3.2. The performance metrics represent the required extra features for tuning the coordination. The structural meta-data information is used to identify component location in a structured application for which this information is extracted. Appendix C, sectionC.1 represents examples of SKIP-compliant sensor files for applications descriptions that have been introduced in Chapter 5.

The coordination engines in the *Measurement* unit will receive the*sensor information* file. This file identifies each component based on the structural meta-data and analyses each building block component based on the extracted performance metrics to generate a new configuration for controlling parameters. Once a new configuration is generated, it constructs an *actuating*

list by applying the same structural meta-data as the sensor file to propose the new controlling parameters configurations to the application. The new configuration file can be used to modify the controlling parameters values of the instrumented RISC-Pb^{2}l building block presented in Listing 3.2.

3.5 Summary

Separating the computational functions from semantic behaviour for each building block, SKIP instruments each component with extra functional requirements and represents its semantic behaviour, regardless of the computational function it delivers. Such a separation provides the ability to interact with each building block in a structured programme and to change the component coordination without altering its computation.

Being a text-based protocol makes SKIP a language independent protocol and grants the flexibility of using a different language for generating an application from the same SKIP compliant structural object in different languages. With the assistance of SKIP adaptor, a SKIP compliant descriptive object for a structured application can be used as a front end language and translated to different parallel frameworks (which support the building block concept) as a back-end parallel language.

Moreover, communication through SKIP objects provides the generalised abstraction between the application logic and coordination functions. This will allow cross-language portability where the application written in different languages communicates with the coordination methods in different languages.

The extensibility of structural parallel programming through RISC-pb^{2}l over heterogeneous multi-core architectures demonstrates the applicability of a building block based approach for targeting the advancement in hardware technologies.

The separation of concern provided by distinguishing the computation from coordination represents implicit and automatic communication and synchronisation through providing the compositions represented in Listing 3.1. Such an abstraction provided through a set of predefined compositions assists programmers to exploit the latest developments in heterogeneous multi-core/many-core architectures with the main focus on computational parts rather than the coordination.

Introducing the ability to interact between a set of coordination engines and RISC-pb^{2}l based structural applications through the SKIP methodology provides the applicability of autonomic coordination, both statically and dynamically.

In the next chapter we implement the proposed SKIP-compliant autonomic behavioural framework to demonstrate the applicability of our SKIP methodology for generating and autonomic coordinating RISC-pb^{2}l structured applications without interfering with and interrupting

the computation functions of the building blocks.

Moreover, our evaluations of the proposed SKIP-compliant autonomic behavioural framework in Chapter 5 and Chapter 6 state that by using SKIP compliant RISC-pb^{2}l structured programming, we can generate structured programmes and interact with its components to modify the coordination on a building block level without interfering with and interrupting the computation functions of the building blocks.

Chapter 4

Performance Enhancement Infrastructure

In this chapter we have developed In this chapter we develop the SKIP compliant autonomic behavioural framework—called Performance Enhancement Infrastructure (PEI)— first presented in Chapter 3, section 3.4. We have embedded FastFlow, a parallel computing framework, that implements the RISC-pb^{2}l building blocks to generate a structured parallel programming. We have extended FastFlow with the extra building block introduced in the SKIP methodology to support autonomic coordination optimisation over heterogeneous architectures. We call it HFastFlow. Moreover, we have instrumented the HFastFlow framework with a set of controllable parameters for the coordination of applications over heterogeneous multi-core architectures. Furthermore, we have provided an adaptor, which automatically generates a structured programme by translating a SKIP compliant object to a HFastFlow application. To demonstrate the interaction ability of SKIP, we have incorporated coordination engines that are capable of autonomously controlling and optimising the coordination conditions presented in Chapter 3, section 3.1, Figure 3.1. The architectural view of a SKIP compliant autonomic behavioural system is composed of the four following parts as demonstrated in Chapter 3, section 3.4, Figure 3.1:

- A set of instrumented SKIP compliant building block libraries.
- A SKIP adaptor to construct the application regarding the provided computational function and supported building block compositions.
- A set of SKIP compliant coordination engines to deliver the coordination decision.
- A SKIP compliant communication mechanism as an interaction bridge between applications and coordination engines.

Where a section is dedicated to explain each part in detail.

A complete product of this research, PEI, is an open source Framework freely downloadable from: https://github.com/mehdi-goli/MC-FastFlow-PEI.

PEI is currently used by a number of paraphrase partners and other external organisations.

In chapter 6 we will present 3 applications (Eispack, MD, and SMTWTP) generated by paraphrase partners that use PEI for performance tuning. Moreover, in chapter 7, section7.2, we will demonstrate the impact of our research illustrated by the adoption of PEI and HFastFlow by different paraphrase partners. Moreover, the proposed GPU back-end presented in section 4.1.1 has been officially accepted as FastFlow GPU back-end [84].

4.1 FastFlow Expansions

Implementing a RISC-pb^{2}l building block library through skeleton patterns, FastFlow provides a set of parallel patterns that is supported by Listing 3.1. We have employed FastFlow to construct and generate the RISC-pb^{2}l based structural parallel programming. Different features have been added to the FastFlow framework, enabling it to execute and automatically tune an application over heterogeneous architecture. Using object oriented concepts, the implementation of the extension has no effect on the existing application. These extensions are in the form of classes that add extra functionality to the FastFlow. Figure 4.1 demonstrates the class diagram that represents extra features added to the FastFlow. As stated in the figure, the classes represented in white are those added to the existing classes (grey) to deliver extra features on FastFlow. The UML notation has been applied to represent the types of class and the relationship between them. In this section we explain the new extended features we added the FastFlow building block framework.

4.1.1 OpenCL Back-end

We have introduced «||code||» by adding an OpenCL back-end to FastFlow [85] to support the *HWrapper* in OpenCl-enabled devices. The FastFlow framework with OpenCL back-end is called HFastFlow. The proposed back-end extends FastFlow's existing parallel patterns (farm and pipeline) in hybrid GPU/CPU multi-core environments. The OpenCL back-end is composed of three components:

- `OpenCLSetUp`: Specifies the required `OpenCLObject` for a given OpenCL device;
- `OpenCLNode`: Encapsulates the OpenCL kernel execution commands inside the FastFlow framework; and,
- `DeviceController`: Provides an instance of OpenCL node for the specified device.

An `OpenCLObject` is composed of the following parameters:

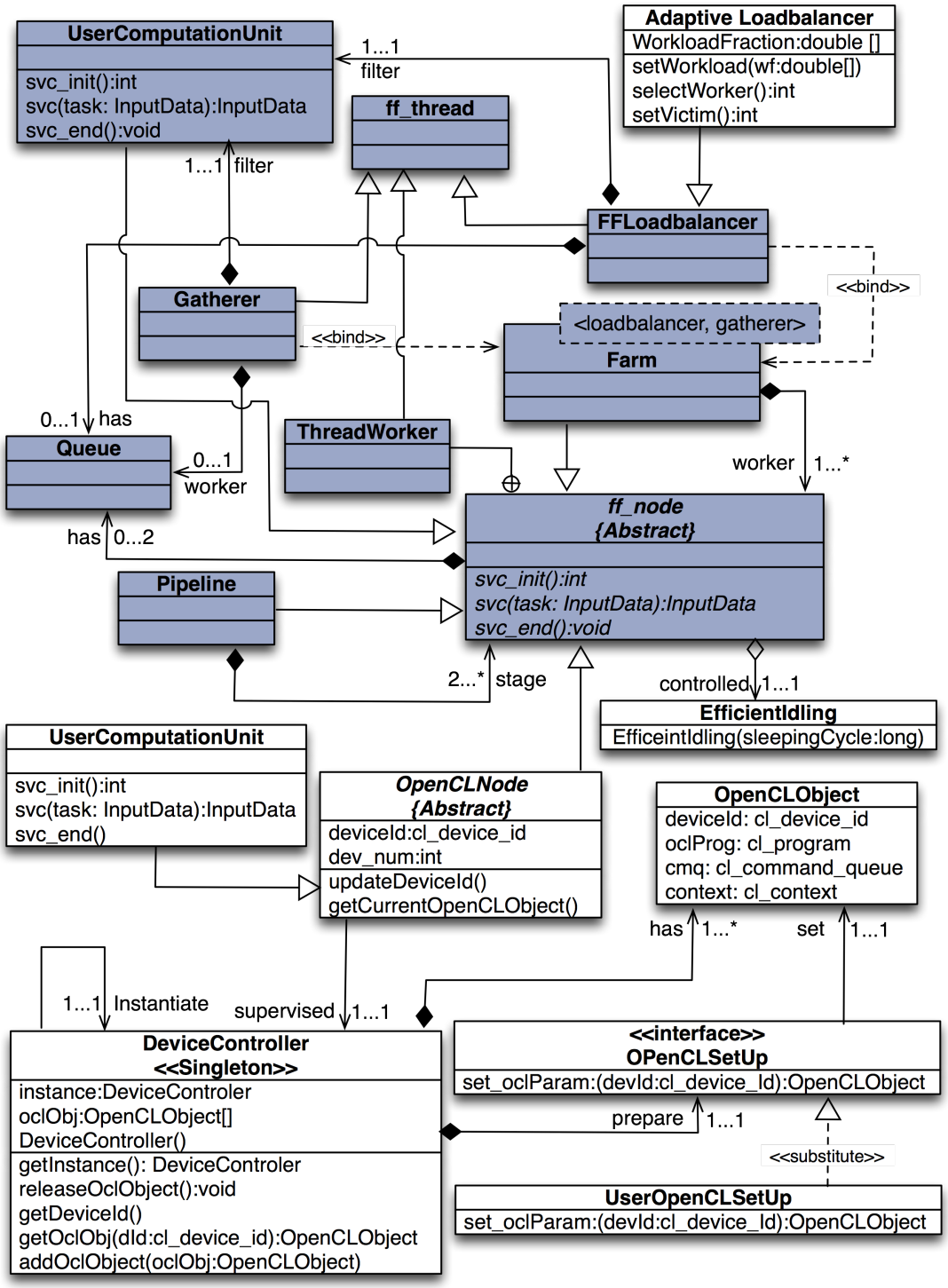


Figure 4.1: A UML class diagram for instrumented FastFlow. Existing classes in FastFlow have been marked in grey

- `deviceId`: Is a unique ID allocated to each OpenCL device. For each OpenCL device, its `deviceId` is extracted by the `DeviceController` and sent as an input parameter to the `set_oclParameter` method.

- `context`: Is used by the OpenCL runtime for managing objects such as command-queues, memory, programme and kernel objects and for executing kernels on one or more devices specified in the context.
- `oclProg`: Represents an OpenCL programme executable from the programme source or binary code.
- `Cmq`: Represents an openCL command queue. It provides a queueing mechanism for executing a kernel on the specified device. It can run the kernels in order or out of order depending on the provided queue specification.

`OPenCLSetUp` Is an interface class containing the `set_oclParameter` method which must be implemented to set the encapsulated `OpenCLObject`.

The user implemented version of the `OPenCLSetUp` class is substituted at runtime. With the substitution, it is possible to arbitrarily customise the `OpenCLObject`.

For each application, there is only one instance of `OPenCLSetUp` per device, despite the number of available OpenCL components.

The method `set_oclParameter` is invoked by the `DeviceController` for all available OpenCL devices.

`OpenCLNode` Is an abstract class, called `ff_oclNode`, derived from `ff_node`. The `svc` function contains the executing process of the OpenCL kernel and it provides access to the determined `OpenCLObject` via the `get_oclParameter` method.

Invoking `get_oclParameter` for an instance of `OpenCLNode` returns the specified instance of an `OpenCLObject` according to the allocated `deviceId` for the instance of `OpenCLNode`. Each `OpenCLNode` has an attribute called `dev_num` which is an integer value that provides the device number assigned to this OpenCL node. The value of the `dev_num` is modifiable. Modifying `dev_num` automatically triggers the `updateDeviceId` method to update `deviceId` to its new value. Therefore, the OpenCL back-end can dynamically migrate from one device to another on demand.

Any class derived from `ff_oclNode` may be incorporated as either a generic pipeline stage or a worker in a farm encapsulating the OpenCL kernel execution command.

`DeviceController` Is a single-tone class responsible for creating and maintaining an instance of OpenCL object per available OpenCL device for an application. It returns the OpenCL object for the defined device through `dev_num`.

OpenCL Component Tuning Depending on the type of OpenCL capable device, tuning the number of threads that execute a kernel over the device is required in order to optimise the

application throughput.

When an OpenCL component is allocated to a GPU device, a large number of threads are required to properly utilise all the stream cores of the GPU device. This is the winning point of a GPU device over a CPU device for an OpenCL component. Usually, a CPU device runs on a much higher clock frequency than the a GPU device. Also, it has large caches which reduce the global access latency. However, a GPU device has a large number of cores which can execute instruction together and wave front scheduling [86] is used to cover up the global access latencies as caches are not very effective in many cases. So the more threads that are created, the greater the level of parallelism and the better the performance will be [44].

When an OpenCL component is allocated on a multi-core CPU device each work-group is more like a task in a thread pool API (like TBB [57] thread pool API). When the size of the work-group is greater than the available number of cores, a CPU thread pool with one thread per core will be created. Those threads will pull work-groups from the task queue appropriately. Therefore, the operating system creates threads that process a large number of work-groups. Each CPU core processes one work-group at a time. Therefore, the group-size has a significant effect on resource utilisation and performance optimisation [44].

However, the case would be slightly different for a building block based structured application with different CPU based and OpenCL based components executing on a heterogeneous multi-core architecture. In this case, if the work-group size is bigger than the number of CPU-cores in the system, all cores will be consumed and other CPU based components of the application will starve as a result of lack of resources. In this case, for an OpenCL component performing on a multi-core CPU device, the work-group size must be controlled manually and adjusted to a value smaller than the number of CPU cores in the system. Therefore, although the same OpenCL kernel can be executed by different workers of a farm, the work-group size for each worker should vary depending on the type of allocated device.

To control the work-group size OpenCL provides two parameters: `global_thread` and `local_thread` [44]. By modifying these two parameters, it is possible to tune the work-group size to optimise the application performance on different OpenCL capable devices.

4.1.2 Adaptive Load-balancer

Moreover, depending on the type of allocated device, an OpenCL farm worker can be either a CPU worker or a GPU worker. Therefore, a farm pattern can be composed of a combination of CPU and GPU workers. When there is a combination of CPU and GPU workers, the service time of a farm worker executing a task can be different.

Here, although the computation performed by each worker is the same, the heterogeneity of the devices creates a significant difference in the service time of CPU and GPU workers. Therefore, the workload distribution of the input stream to each worker should be in proportion

with their service time.

To deal with this issue, we provide a weight-based load-balancer called an `adaptive_loadbalancer`. The `adaptive_loadbalancer` subclasses the Round-Robin `ff_loadbalancer` in HFastFlow farms.

The `adaptive_loadbalancer` biases the workload distribution with regards to the provided workload fraction for farm workers.

4.1.3 Memory Management

The queue system in the HFastFlow framework can be *bound* or *unbound*. A bound queue has a fixed size that limits the maximum number of tasks they can buffer simultaneously, while an unbound queue can accommodate an unlimited number of tasks as long as the system memory allows this. the bound queue is very simple, elegant and performs very well on multi-cores when the producer and the consumer work on different cache lines [87].

In HFastFlow the queue is internal, which meanes that each component reads/writes from/to its private queue with no interference from other components. Therefore, for patterns like farm, it is the responsibility of a load-balancer to distribute the data among worker components [87]. In this case, for a farm with OpenCL workers that are capable of being executed on different devices, a bound queue can prevent the slow workers in a farm from being overloaded.

However, unbound queues are more general than bound queues. They are mostly preferred to avoid deadlock issues without introducing heavy communication protocols in the case of complex streaming networks, i.e. a graph with multiple nested cycles. It may not be easy to determine the correct `queuesize` for the bound queue and unless a learning mechanism is involved, it needs to be set up for the worst case [87].

Despite the benefit of unbound queues, there are situation where it is necessary to bind the queues, especially when high-throughput, coarse-grained and resource intensive workload application with significant memory demand is available. It is also critical to control the memory usage in a system especially for applications with large-scale input data and heterogeneous components with different computing powers in order to avoid a system crash.

In HFastFlow, we can only bind the input queues of computing components statically. However, exhausting all queue slots for computing components can vary dynamically depending on the computing power of each component, as the computing power of a component can change depending on the type of the allocated device. By adaptively controlling the memory usage of components, we can use the unbound queue and throttle the feeder of computing component to avoid a system crash. The feeder of a computing component for a Farm pattern is the load-balancer and for the pipeline pattern the feeder for stage i is stage $i-1$. Having adaptive memory usage controller is particularly significant in streaming applications where processing occurs outside HFastFlow and it is necessary to provide sufficient memory for other processes.

With minimal modifications to the underlying framework, we have introduced adaptive throttling to regulate the memory footprint of the application and maintain usage within configurable bounds. A computing component is starved of input when the available free memory falls below the STOP_THRESHOLD as a percentage of total memory. Processing on the other stages proceeds until memory usage rises above the START_THRESHOLD, at this point, the feeder thread is awakened and new input is injected into the component. The conceptual simplicity and effectiveness of this approach makes it an attractive solution to the memory management problem. Processing continues as other parallel pipeline stages make use of the available hardware, regardless of the actual location of the bottleneck stage.

4.1.4 Efficient Idling

The key challenge here is to obtain maximum utilisation of the resources for any CPU slot allocated to a component. The optimised usage of the allocated slot for each component depends on the availability of a task in the component queue. If the task is not available, the component must wait. This waiting can be either due to a busy waiting loop or a thread sleeping until the data becomes available. In the former especially when the number of the threads running the application is less than or equal to the number of available cores, the performance of the application does not drop. However, when the number of threads running the application is greater than the number of cores, by using the sleeping mechanism this time slot can be dedicated to those threads that have received their tasks but are waiting for resources to execute them. Moreover, the energy consumption of the former case is higher and less optimised than the latter, which is also against the optimal resource utilisation.

Therefore, we provide an extra information layer on top of the operating system scheduler by putting a thread which is executing a component to sleep whenever the component input/output queue is empty/full.

Applying such a strategy in frameworks that support busy waiting mechanisms can be made on demand and is useful when the number of threads is greater than the number of available cores and also energy saving and optimal utilisation are of high priority.

Therefore, a tunable parameter `Idling_Status` has been considered for selecting between the two strategies. The actuating state of the `Idling_Status` parameter can be set via an actuator that receives a constraint configuration to demonstrate the actuating state.

We provide a boolean flag `FF_SAVER` as a constraint parameter to determine the applicability of the efficient idling policy. By default the flag is set to false, meaning the efficient idling policy is not applied.

4.2 HFastFlow Instrumentation

HFastFlow is an implementation of extended RISC-Pb^{2}l building block library that is presented in Chapter 2, Table 2.3. We instrument HFastFlow building blocks by adding controlling parameters (presented in Chapter 3, Table 3.3), performance metrics (presented in Chapter 3, Table 3.4), and structural meta-data (presented in Chapter 3, Table 3.2) These instrumentations are used to generate SKIP-compliant building block components.

4.2.1 Controlling Parameters

Table 4.1 represents a set of controlling parameters added to HFastFlow components (presented in Figure 4.1) to generate SKIP-compliant HFastFlow components. The first column represents the controlling parameters; the second column represents a list of HFastFlow components containing these parameters; and the third column represents the parameter definition in HFastFlow. These controlling parameters are also called *actuators*.

4.2.2 Performance Metrics

Table 4.2 represents a set of performance metrics added to the HFastFlow components (presented in Figure 4.1) in order to generate SKIP-compliant HFastFlow components. The first column represents the performance metrics; the second column represents a list of HFastFlow components containing these metrics; and the third column represents the definition of each metric. These performance metrics are also called *sensors* and are extracted at runtime. Each node in HFastFlow periodically updates its parameters. The updating interval is determined by the sampling mode factor.

4.2.3 Structural Meta-data

Table 4.3 represents a set of structural meta-data added to the HFastFlow components (presented in Figure 4.1) in order to generate SKIP-compliant HFastFlow components. The first column represents the structural meta-data; the second column represents a list of HFastFlow components containing these parameters; and the third column represents the definition of each parameter.

4.3 High-Level Abstraction Layer (HAL)

HAL is composed of *i)* a user-level abstraction layer—called *SKIP Adaptor*— that constructs a structural application by receiving a SKIP-compliant descriptive definition of the application from a user; *ii)* a system-level abstraction layer—called *OpenCL Device Virtualisation Layer (ODVL)*— that unify OpenCL-enabled devices on a Heterogeneous multi-core systems; and *iii)*

Table 4.1: Control parameters

Controlling parameter	HFastFlow Instrumented Components	Definition
WORKLOAD	ff_loadbalancer and adaptive_loadbalancer	Is a vector of floating point values that determines the workload fraction for each worker of a farm pattern. It is possible to readjust the workload distribution with a new workload value both statically and dynamically.
APPLICATION_STATUS	ff_loadbalancer,ff_gatherer, adaptive_loadbalancer, ff_node, oclNode, ff_farm and ff_pipeline	Is a string variable that determines the status of a structured application. Its value can either be SUSPEND, meaning that an application is suspended or RESUME, meaning that an application is running. Once this is set for an application, its value is propagated to all components.
PRIORITY	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode ff_farm and ff_pipeline	Is an integer variable that determines the application priority in the system. Once this is set for an application, its value is propagated to all components.
CH_BOUND	ff_loadbalancer,ff_gatherer, adaptive_loadbalancer, ff_node, oclNode,	Is an integer variable that determines the maximum number of tasks a component can buffer in an input/output queue. There is no limit on the default queue size for each component.
PROCESSOR_NUMBER	ff_loadbalancer,ff_gatherer, adaptive_loadbalancer, ff_node, oclNode	Is an integer variable to determine the processor number for a component.
DEVICE_NUMBER	oclNode	Is an integer variable to represent the device number for an OpenCL component.
DEVICE_TYPE	oclNode	Is a string variable to determine the device type for an OpenCL component.
COMPONENT_STATUS	ff_loadbalancer,ff_gatherer, adaptive_loadbalancer, ff_node, oclNode,	Is a boolean to determine the status of a component in the system which can be either *ON* or *OFF*. The former means that the component is active, while the latter represents that the component has been suspended.
MASKING	ff_node, oclNode,	Is a vector of boolean values that represents whether or not a worker of a Farm pattern is active to receive a task from a load-balancer.
PROFILE_AND_TUNE	ff_farm and ff_pipeline	Is a boolean variable representing whether or not the coordination mechanism should be imposed on an application.

Table 4.2: Performance metrics.

Performance Metrics	HFastFlow Instrumented Components	Definition
PROCESSED_TASKS	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a long integer variable that determines the total number of tasks processed so far by this component.
Queue_INPUT	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a long integer variable that determines the number of tasks inside the input queue of a component yet to be processed.
Queue_OUTPUT	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a long integer variable that determines the number of tasks inside the output queue of a component processed by the component.
Queue_LIMIT	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is an integer variable that represents the maximum number of tasks for a component that can be buffered for processing.
COMPONENT_LAST_PROCESSING_TIME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode	Is a floating point variable that represents the service time of the last execution.
COMPONENT_DISTRIBUTION_TIME	ff_node and oclNode	Is a vector of floating point value that represents a list of service time frequency distributions during component execution.
TOTAL_COMPONENT_ACTIVE_TIME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a floating point variable that represents the total service time of a component on which the component was executing a task.
ELAPSED_TIME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode ff_farm, and ff_pipeline	Is a floating point variable that represents the total executiontime of a component since the beginning of the execution of an application.
ASSIGNED_DEVICE_NUMBER	oclNode	Is an integer variable to represent the device number that is assigned to an OpenCL component.
END_RECEIVED	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode	Is a boolean variable that represents whether or not a component has exhausted all its allocated tasks.
CH_IN	ff_loadbalancer and adaptive_loadbalancer	Is an integer variable that represents the number of tasks inserted to a load-balancer.
CH_OUT	ff_loadbalancer and adaptive_loadbalancer	Is an integer variable to represent the number of tasks sent by a load-balancer.
POP_DELAY_TIME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a floating point variable that represents the total waiting time of a component for an empty input queue.
PUSH_DELAY_TIME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a floating point variable that represents the total waiting time of a component for a full output queue.
POP_DELAY_COUNT	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a long integer variable that represents the total number of times a component waits for an empty input queue.
PUSH_DELAY_COUNT	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, and oclNode	Is a floating point variable that represents the total number of times a component has to wait for a full output queue.

Table 4.3: Structural meta-data

Structural Meta-data	HFastFlow Instrumented Components	Definition
APPLICATION_NAME	ff_farm and ff_pipeline	Is a string variable that determines the name of the application.
COMPONENT_NAME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode ff_farm, and ff_pipeline	Is a string variable that determines the name of a component.
STRUCTURE_TYPE	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode ff_farm, and ff_pipeline	Is a string variable that determines the type of a component.
NODE_ADDRESS	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, ff_node, oclNode ff_farm, and ff_pipeline	Is a string variable that determines a URL address of a node in the directed graph of a structured application.
FUNCTION_NAME	ff_loadbalancer, ff_gatherer, adaptive_loadbalancer, and ff_node	Is a string variable that represents the name of the computational function executed by a component.
CH_POLICY	ff_loadbalancer, ff_gatherer, and adaptive_loadbalancer	Is a string variable that represents the policies provided for a load-balancer or gatherer.
CH_MULTIPLICITY	ff_loadbalancer, ff_gatherer, and adaptive_loadbalancer	Is an integer variable that represents the multiplicity of a load-balancer or gatherer.
FEEDBACK	ff_farm and ff_pipeline	Is a boolean variable that represents whether or not the component should reroute the output data relative to input data.
PAR_LEVEL	ff_farm	Is an integer variable to determine the number of identical components to be executed in parallel.
SETUP_FUNCTION_NAME	oclNode	Is a string variable that represents the set up function name required to prepare the device building block in the host site.
KERNEL_PATH	oclNode	Is a string variable that represents the URL in the location of the kernel file executed on the device.

an autonomic manager—called *Dynamic Skeleton Runtime interface (DSRI)*— that coordinate structural applications over heterogeneous multi-core architecture.

4.3.1 SKIP Adaptor

Using SKIP compliant structured programming, the SKIP adaptor provides a clean separation between the coordination and computation functions as stated in Chapter 3, Section 3.4.

The adaptor receives a descriptive SKIP object to determine the building blocks compositions of an application demonstrated by a set of structural meta-data instructions. It also receives a library address that contains the predefined computational functions executed by different building blocks By receiving these two files, SKIP adaptor automatically constructs a structural application as defined in the SKIP methodology represented in Chapter 3, Listing 3.2.

Each building block based framework must provide its own SKIP adaptor to translate the SKIP compliant object into a structured application executing on that framework.

In the following we explain the different parts of the SKIP adaptor implemented for the HFastFlow framework. Although the composition rules proposed in the SKIP methodology is capable of generating and instrumenting all parallel patterns that is supported by RISC-pb^{2}l as stated in Appendix A, the operability of these patterns depends on their implementation through the HFastFlow framework. Currently HFastFlow only implements the RISC-pb^{2}l compositions that is required for farm and pipeline patterns. Therefore, full operability of all the compositional features of RISC-pb^{2}l libraries will be available when these compositions are implemented by the HFastFlow framework. Here, the SKIP adaptor for HFastFlow is confined to generating applications that are only designed by the RISC-pb^{2}l compositions for farm and pipeline patterns.

4.3.1.1 Computational Function Type

The SKIP adaptor will operate on a different pre-defined computational function. Each Framework has its own computation function signatures as stated in Chapter 3, Section 3.4.

In HFastFlow, any computational function generated for the building blocks must have one of the following signatures:

- `void* func(void*)` for any `ff_node` function.
- `oclParameter* setPar(cl_device_id id)` for setting the OpenCL parameter for an OpenCL node.
- `void* oclfunc(void*,oclParameter*)` for any OpenCL execution function.

4.3.1.2 Structural Factory

A SKIP compliant object represents a structural tree for an application which satisfies the composition rules presented in Chapter 3, Section 3.3, Listing 3.2.

`structuralFactory` will receive the SKIP-compliant object representing the structural composition of a HFastFlow application. Appendix B represents the structural composition of HFastFlow applications used in this thesis for PEI evaluation. The `structuralFactory` method recursively parses the structural meta-data to construct the application from the SKIP structured tree object. It uses different functions to construct different HFastFlow components.

Table 4.4 represents the functions used by `structuralFactory` to construct the HFastFlow component. The first Column in the table represents the function name; the second column in the table states the RISC-pb^{2}l building block that is generated by that function; and The Third column in the table represents the HFastFlow component that is instantiated by that function to build SKIP-compliant HFastFlow application.

`structuralFactory` parses the SKIP-compliant object received as an input, for each RISC-Pb^{2}l building block in the SKIP-compliant object file. It then invokes its corresponding function represented in the first column of the Table 4.4 and constructs the corresponding HFastFlow component for that building block represented in the third column of the Table 4.4. Once a component is constructed, it will be assembled in the appropriate location in the application structural tree corresponding to the given SKIP structural meta-data.

Each building block has a unique address given by `structuralFactory` according to their url position in the structural tree. This address is available via `node_name`. Therefore, each component is not only accessible via processing the structured tree but also reachable by direct access through its global `node_name`. This will eliminate the overhead of accessing an individual component.

4.3.2 Dynamic Structural Runtime Interface

The dynamic structural runtime interface (DSRI) is used as a bridge for communicating between the coordination engines and a structural framework like HFastFlow. The DSRI architecture is composed of a DSRI client and a DSRI server connected by an asynchronous messaging queue.

The DSRI client and server provide both intra-process and inter-process communication between the HFastFlow framework and coordination engines. Depending on the location of the coordination methods, either communication method could be selected.

- *Inter Process Communication*: It is used when the two methods calling each other are in two separate processes. The communication can be provided via a remote procedure call (RPC) [88].

Table 4.4: SKIP adaptor functions to generate the HFastFlow structured application from structural meta-data instruction

Function name	Function Definition	HFastFlow Instantiated Component
`buildFarm`	This function is responsible for mapping the 'parallel' ':' '[' ' { ' 'parLevel' ':' ' x' ' , ' * ' } ' ']' or 'MISD' ':' '[' ' { ' $*_1$ ',' ... ',' $*_x$ ' } ' ']' into the farm pattern provided by HFastFlow.	`ff_farm`
`buildEmitter`	This function maps the 'spread' ':' ' { ' 'CH_POLICY' ':' 'STRING' ',' 'CH_MULTIPLICITY' ':' INT ',' 'WORKLOAD' ':' '[' 'VALS' ']' ',' SEQ ' } ' into its corresponding `emitter` component provided by HFastFlow.	`ff_node`
`buildLoadbalancer`	This function maps the 'd-pol' ':' ' { ' 'CH_POLICY' ':' 'STRING' ',' 'CH_MULTIPLICITY' ':' INT ',' SEQ ' } ' into its corresponding `ff_loadbalancer` component provided by HFastFlow.	`ff_loadbalancer` or `adaptive_loadbalancer`
`buildCollector`	This function maps the 'reduce' ':' ' { ' 'CH_POLICY' ':' 'STRING' ',' 'CH_MULTIPLICITY' ':' INT ',' SEQ ' } ' into its corresponding `collector` component provided by HFastFlow.	`ff_node`
`buildGatherer`	This function maps the 'g-pol' ':' ' { ' 'CH_POLICY' ':' 'STRING' ',' 'CH_MULTIPLICITY' ':' INT ',' SEQ ' } ' into its corresponding `ff_gatherer` component provided by HFastFlow.	`ff_gatherer`
`buildHSEQ`	Generates an `ff_oclnode` component encapsulating the provided OpenCL based computation function in its `svc` body.	`ff_oclNode`
`buidSEQ`	Generates an `ff_node` component encapsulating the provided sequential computation function in its `svc` body.	`ff_node`
`setupFunction`	Replaces the provided *setupfunction* with the `OpenCLSetUp` interface function to specify the required `OpenCLobject` for a given OpenCL device on the host site.	`UserOpenCLSetup`
`buildPipe`	Maps a set of SKIP objects separated by ',' in to the HFastFlow pipeline pattern when the specified structure type for their parents is set to pipeline.	`ff_pipeline`
`addFeedback`	Applies the HFastFlow `wrap_around()` method provided for the farm and pipeline to implement the feedback for the SKIP object.	`ff_farm` and `ff_pipeline` with `wrap_around()` method

- *Intra Process Communication*: It is used when the coordination methods are a function within the application method. This communication can be provided by a direct function call.

The information flowing in both inter and intra process communications are in the form of SKIP compliant objects.

In our case, inter process communication is processed by *ZeroMq* as an asynchronous queueing system to transfer the SKIP Object. *ZeroMq* is a very lightweight messaging system specially designed for high throughput/low latency scenarios. It supports many advanced messaging scenarios by combining and implementing various pieces of the framework (e.g., sockets and devices). This will make *ZeroMq* a very flexible queueing system [89].

4.3.2.1 DSRI Client

The DSRI client is an independent node embedded inside the HFastFlow framework. It is executed in a separate thread. It uses a single mutual exclusion technique during the information exchange to minimise the performance overhead of the system.

Using SKIP, the DSRI client is a trader between the DSRI server and an executing application. It provides access to retrieve the sensors from the instrumented components of an application and sends them to the DSRI server and also forwards the instructions from the DSRI server to the application components.

Figure 4.2 demonstrates the class diagram that represents the DSRI client and its relationship with the SKIP adaptor and HFastFlow application.

Client Manager The client manager is the coordinator of the DSRI client. The main responsibility of the client manager is to extract the sensor information and to deliver the dynamic coordination decisions to the actuators. Towards this aim, client Manager provides two interfaces `getter` and `setter`, that each instrumented framework such as HFastFlow must implement. In the following we explain the implementation of these two interfaces in HFastFlow.

As an application structure has a tree format all SKIP compliant actuator and sensor objects also have a tree format. The root node is always a point of interaction between DSRI client and a HFastFlow application. Using the two methods, the client manager will recursively retrieve/modify the sensors/actuators from the root to any terminal node in the tree structure of a SKIP compliant object provided for an application.

Implementation of `getter` and `setter` Interfaces In the instrumented version of HFastFlow, all terminal nodes(i.e. `ff_node`, `ff_oclNode`, `ff_loadbalancer`, `ff_gatherer` and `adaptive_loadbalancer`) and non-terminal nodes (i.e `ff_farm` and `ff_pipeline`) implement the two interfaces `getter` and `setter` to interact with DSRI client. For each HFastFlow node used in an application structure, the `getter` retrieves its sensor information encodes it into a SKIP compliant object. The generated SKIP compliant object for each node is a collection of performance metrics entangled with the structural meta-data related to that node. The structural meta-data information provides a unique reference to the structured application's (sub)-tree nodes.

The non-terminal nodes also recursively retrieve the SKIP compliant object from their children and encapsulate them with their SKIP compliant object to form the composition they are representing. Therefore, the root node in an application contains a SKIP compliant object for the whole application.

Similar to the `getter` function each `setter` delivers the coordination decisions to the provided actuators for its node. It receives a SKIP compliant object, decodes it and adjusts the

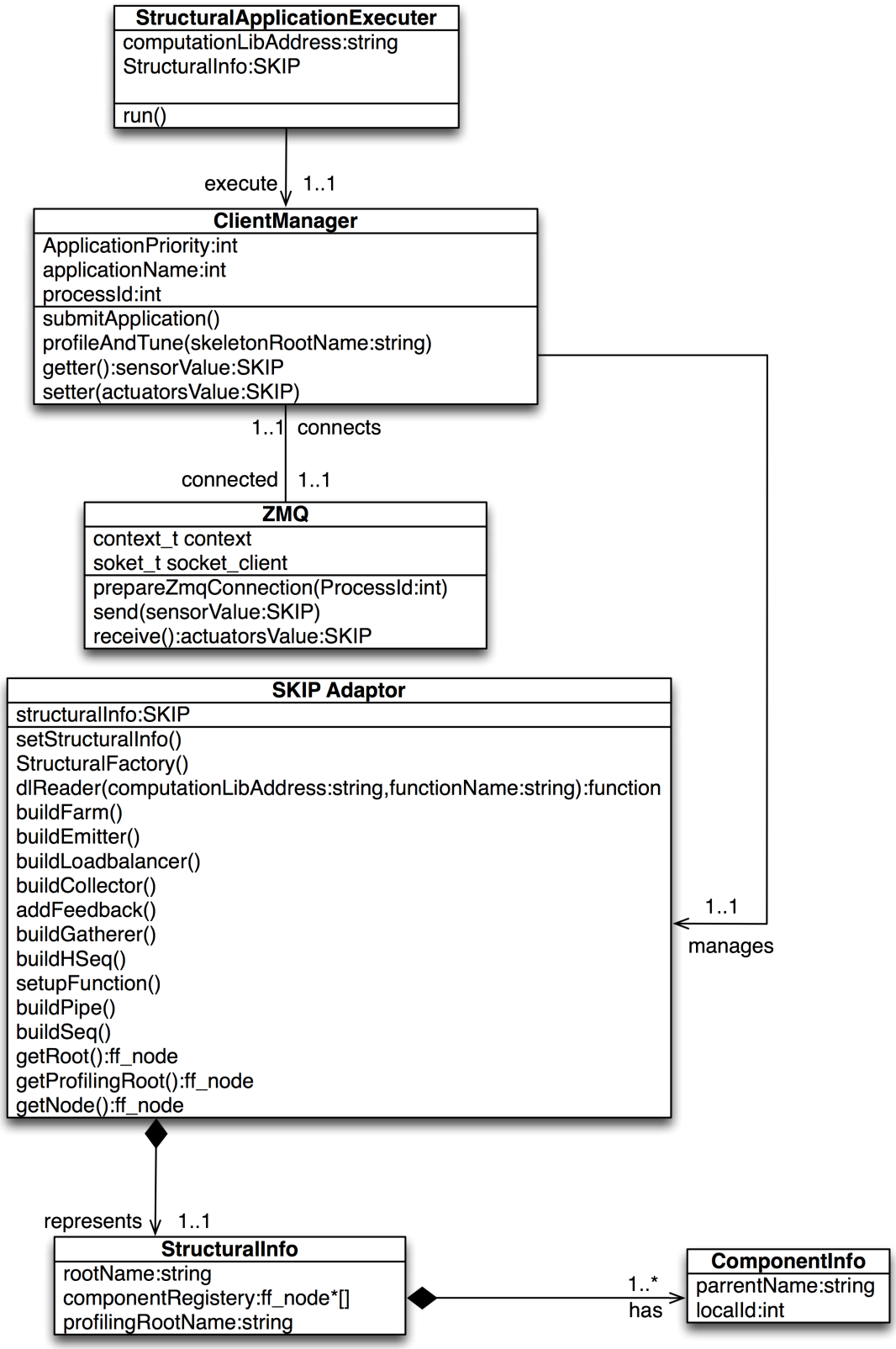


Figure 4.2: A class diagram representing the DSRI client and its relationship with client side objects

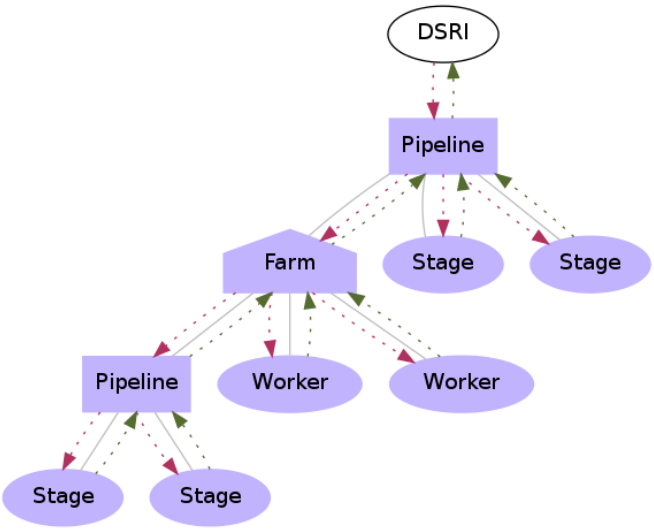


Figure 4.3: SKIP-compliant information exchange between DSRI and a sample HFastFlow application.

actuator states for its related pattern.

Similar to all SKIP compliant objects, the actuator file also has a tree structure format. Therefore, the root of the tree contains all actuators' information provided for all nodes of an application. The non-terminal nodes also recursively dispatch the SKIP compliant object to their children.

Figure 4.3 represents the information exchange between an application in a instrumented HFastFlow and DSRI client. DSRI client extract the sensor information via getter interface (green arrows) and to deliver the dynamic coordination decisions via setter interface (red arrows) to the actuators.

Structural Application Executor A structural application executor is the port for executing an application. It instantiates the client manager, which generates an instance of DSRI client for the given application. The client manager invokes the SKIP adaptor to generate the application from the given SKIP object in the instrumented HFastFlow framework. The `run` method in the structural application executor executes the application. Once executed, the provided client manager registers the application to the DSRI server for further profiling and dynamic tuning.

4.3.2.2 DSRI Server

Figure 4.4 shows the class digram for the DSRI server and its relationship to the provided coordination engines. The DSRI server, is responsible for exchanging the SKIP compliant objects between the DSRI client and the coordination engines. It executes in a different process from applications.

The DSRI server incorporates some coordination methods or act as a proxy for SKIP compliant external methods that provide coordination. Unlike the DSRI client there is only one DSRI server per machine node.

Server Manager For each application, the server manager receives a SKIP compliant sensor object, in each time interval, from the DSRI client of an application. It dispatches the received object to the coordination methods and retrieves the appropriate SKIP compliant actuator objects from the coordination methods. It then sends the SKIP compliant actuator objects to the DSRI client, as a corresponding response to the SKIP compliant sensor object of the requested DSRI client.

Also, it normally receives environmental constraints in order to configure the environment for executing applications.

For each application the server manager stores the last SKIP compliant sensor object in a historical database when the application execution has finished. The stored information will provide pre-run knowledge about that application for future execution.

Registering a new application and de-registering an application are performed by the server manager. Once a new application is registered according to *(i)* the submitted SKIP object; *(ii)* availability of any historical data for that application; and *(iii)* the available coordinating methods; new actuating states for the application are constructed.

The process of updating the actuating states for each application is performed dynamically until the application terminates. Depending on an application's nature, the frequency of exchanging information between DSRI and the application can be specified as a constraint object for the application.

DSRI Server Instrumentation DSRI Server has been instrumented with SKIP-compliant constraint configuration that is presented in Chapter 2, section 2.2.2, Table 2.3 for determining environmental changes in the system. Table 4.5 represents a set of constraint configuration added to the DSRI Server. The first column represents the constrain parameter and the second column represents the definition of each parameter. The constraint configuration can be set statically in order to impose certain limitations on both applications and executing environments.

4.3.3 ODVL: OpenCL Device Virtualisation Layer

ODVL provides a virtual representation of all available OpenCL capable devices for the underlying heterogeneous multi-core system. Taking advantage of a dynamic compilation, most OpenCL components, if designed with parametrisable global and local threads and also parametrisable group-size, can be migrateable among OpenCL resources. Therefore, there is no need to

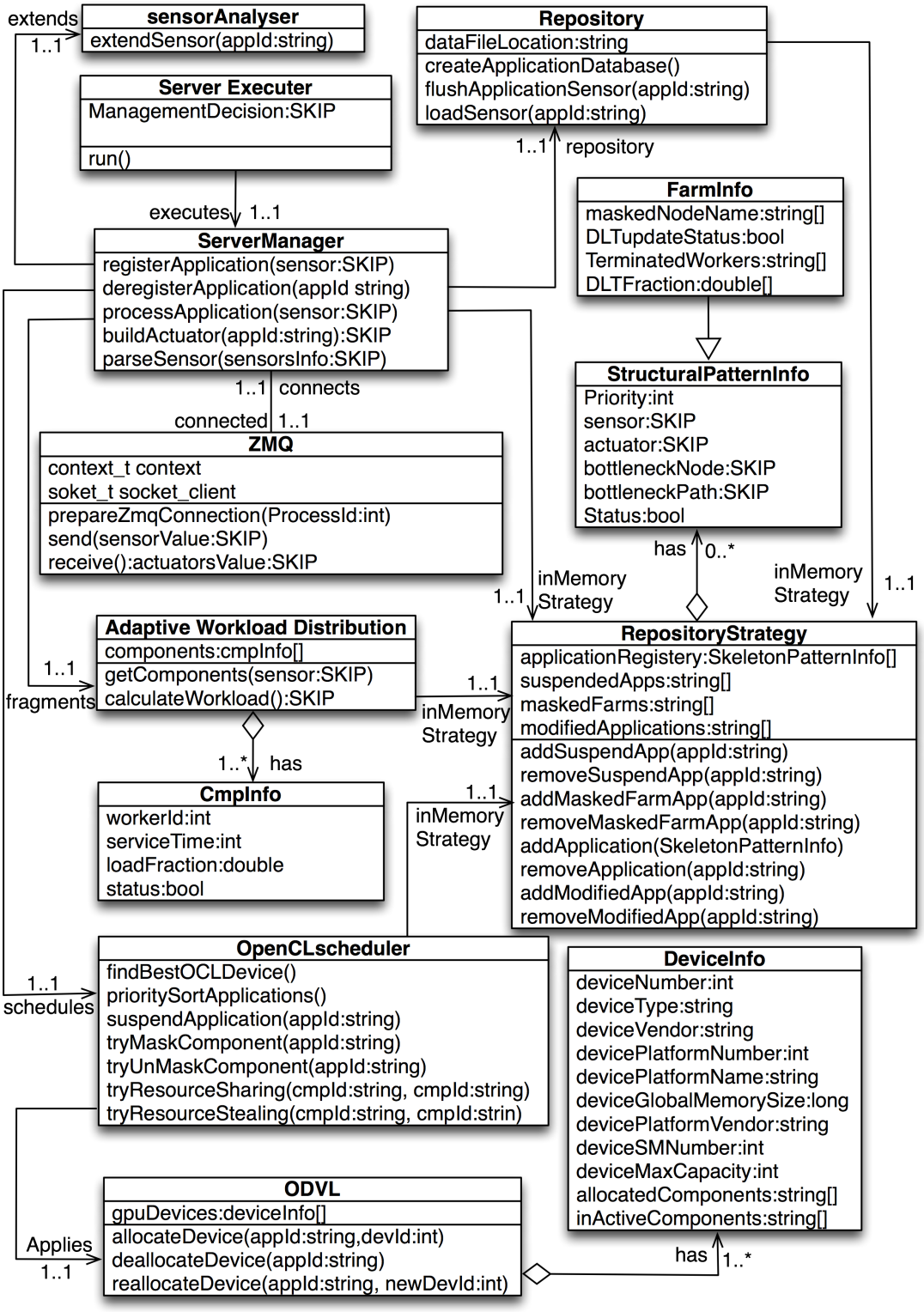


Figure 4.4: A class diagram representing the DSRI Server and its relationship with the embedded coordination engines.

Table 4.5: Constraint configurations added to DSRI Server

Parameter name	Parameter Definition
PRIORITY_POLICY	Is a string variable to define the type of priority used for an application. It can be either a FIXD, meaning that the application priority will not change over time; or VARIABLE, meaning that the application will age over time based on the determined ageing algorithm.
MAXIMUM_BB_PER_DEVICE	Is an integer variable to set the maximum number of components that can be allocated on a device without any drop in performance or causing the the system to crash.
DAMPING_RATIO:	Is a floating point variable that determines the level of sensitivity. The higher the value, the less sensitive the system. However, selecting a number that is too small will overshoot the desired output.
SAMPLING_MODE	Is a string variable to control the level of information monitoring for service time, push delay and pop delay frequency distribution. It can be either AGGRESSIVE where comprehensive information is gathered on every individual task completed by a component; or SPARSE where statistical sampling is performed at a specified sampling rate for only a certain fraction of the completed tasks. This is intended to be the default mode.

provide alternative components that are executed on different devices. This will save a lot of programming effort.

The ODVL contains a set of devices called deviceSet. Each deviceSet entry contains the required information about its representing device. Such information is the device number, the device type, the number of running applications on the device, the maximum number of computing units and the maximum size of memory for that device. Also, each device is equipped with a devCAP parameter that represents the maximum number of applications executed at the same time on a device without any performance drop in allocated applications. The devCAP parameter can be set as an environmental constraint. The ODVL is responsible for allocating, reallocating and deallocating the provided resources for components.

Resource Allocation: Is responsible for allocating the selected resources to a component. Each device represented in a deviceSet is equipped with a set called appSet. Each entry of the appSet presents the information about the allocated application to the device. Such information is the application name and a set called CompSet, which contains a list of component names of the application allocated to this device.

Resource Deallocation: Is invoked when an application terminates. Once invoked, it updates the status of all the resources used by the application. The application entry is removed from all used devices in the deviceSet. Once cleared, invoking the components destructor

frees the actual devices.

Resource Reallocation: Attempts to reallocate the component to a newly selected device. If successful, it returns `true` and modifies the old device entry and the newly selected device entry in the `deviceSet`. Such modifications are the number of running applications and the list of allocated applications. If not successful, it returns false and the component is switched back to the previously selected device.

Although the OpenCL component is removed from the device entry, the component itself is deactivated rather than destroyed. This is due to the process of building an OpenCL programme at runtime being expensive [90]. As reallocation of the same component to the same device can be repeated several times in the system, the rebuild process can have a significant effect on the application's performance. Hence, once an OpenCL component is created for a device it will not be destroyed until the application terminates.

4.4 Performance Enhancement Tools (PETs)

PEI has a list of coordination engines in order to modify the the controlling parameters added to HFastFlow.

The connecting bridge between the coordination engines and instrumented applications is the DSRI server.

Controlling decisions generated by the provided coordination engines are transferred by the DSRI server as a SKIP compliant actuator files to the instrumented application through *ZeroMq* connections.

Embedded coordination methods are entangled with the DSRI server in the same process. The DSRI server invokes these methods through a function call.

The external coordination method provided here is a static structural configuration that uses the DSRI server as a proxy to exchange the information with structured applications. In this section unless it is explicitly mentioned, the coordination methods are embedded.

Our coordination engines include: *i) Sensor Analyser*; *i)) Adaptive workload distribution*; *iii) OpenCL scheduler*; and *iv) Static Structural Configuration*. Other coordination engines can be considered such as engines optimising memory allocation for offloading input data to GPU devices in an application; or engines restructuring components compositions in an application that is known as refactoring. In this thesis, we have designed and developed the following coordination engines as they have been considered as key optimisation objectives for ParaPhrase project [40].

4.4.1 Sensor Analyser

When invoked, our provided `sensorAnalyser` explores a received sensor file and calculates the following performance metrics for each component of it.

- *Queue Throughput*:

$$TQ_i = \frac{queueoutput}{queueinput} \tag{4.1}$$

  where the *queuinput* is the number of input tasks inserted to the queue Q_i; and the *queueoutput* is the number of output tasks consumed from the queue Q_i. This will be useful when an external queueing system is provided.

- *Component Utilisation*:

$$CU = \frac{cActiveTime}{totalTime} \tag{4.2}$$

  where the *cActiveTime* represents the amount of time that a component actively has to process tasks; and *totalTime* shows the amount of time elapsed so far.

- *Component Efficiency Rate*:

$$CER = \frac{TQ_{Ci} + (1 - CU_i)}{2} \tag{4.3}$$

  where the TQ_{Ci} is the throughput of the input queue (TQ) connected to component Ci; and CU_i is the utilisation of component Ci. The greater the component utilisation is and the less the component throughput is, the less the component efficiency is. Dividing it by 2, CER is normalised within [0,1].

Also, the `sensorAnalyser` extracts the Application bottleneck node and bottleneck path for every received sensor object from an application.

The application bottleneck path determines a path in an application tree structure from the root component to the bottleneck component. An application bottleneck node is the component with the minimum CER in a structured application tree.

4.4.2 Adaptive Workload Distribution

In a farm pattern with different OpenCL workers allocated to heterogeneous devices, although the results of all workers are the same, there would be a significant difference in workers' throughput.

A key challenge is to distribute the input streams among the farm workers in proportion with their throughputs. In this case, all workers in a farm terminate simultaneously.

Based on the divisible load theory [91], our adaptive workload distribution calculates the workload distribution that is relevant to the workers' service time.

We define

$$T_{wi} = \alpha_{wi} \times S_{wi} \tag{4.4}$$

where T_{wi} is the total computation time of the assigned tasks to the worker i. α_{wi} is a number between [0,1] representing the fraction of the workload allocated to the worker i. S_{wi} is the service time of a worker to execute a unit of task i.

In order to have all the workers of a farm pattern finish at the same time, in a farm with N workers we would have:

$$T_{w0} = T_{w1} = ... = T_{wN} \tag{4.5}$$

with the following condition:

$$\alpha_{w0} + \alpha_{w1} + ... + \alpha_{wN} = 1 \tag{4.6}$$

To simplify the workload equation for each worker, with no loss of generality, we limit the number of workers to 2. Combining equations 4.4, 4.5 and 4.6 for a Farm with 2 workers, the fraction of the workload for w_0 is:

$$\alpha_{w0} = \frac{S_{w0}}{S_{w0} \times (1 + \frac{S_{w0}}{S_{w1}})} \tag{4.7}$$

and the fraction of the workload for w_1 is:

$$\alpha_{w1} = \frac{S_{w0}}{S_{w1} \times (1 + \frac{S_{w0}}{S_{w1}})} \tag{4.8}$$

Considering $1 = \frac{S_{w0}}{S_{w0}}$, by extending the equation for N workers, the fraction of the workload for w_i is:

$$\alpha_{wi} = \frac{S_{w0}}{S_{wi}(\sum_{i=0}^{i=N} \frac{S_{w0}}{S_{wi}})} \tag{4.9}$$

The workload calculator method receives a sensor object containing the application tree structure. The method recursively calls itself for each component in the sensor object. It generates a SKIP compliant actuator object where

- For a farm component, it contains a workload distribution vector of α_i representing the fraction workload for worker w_i; and
- For other components, the workload distribution value would be null.

4.4.3 OpenCL Scheduler

A scheduler has been implemented to allocate different OpenCL components to available OpenCL devices. Operating on top of the ODVL, our OpenCL scheduler supports multi-tenant application executions, where different applications can be executed simultaneously. If enough resources are provided the applications run in parallel. Otherwise, the applications run concurrently according to the user determined priority policy.

Figure 4.5 indicates the architectural view of the proposed OpenCL scheduler. The OpenCL scheduler is mainly composed of four different types of modules (Priority Manager, Adaptive Allocator, Component Switcher and Masked Component) that operate on four different types of queue (Active Application, Suspended Application, Active Component and Masked Component). When required, a module interacts with the ODVL and application registry layers in order to modify a queue. In the following we explain different parts of the OpenCL scheduler in detail.

4.4.3.1 Priority Based System

Priority Policy: Are determined as *fix* and *variable* policies. If the *fix* policy is applied for an application, the application priority will not change during the execution.

If the *variable* policy is applied for an application, the application will age during its execution.

When the application priority is not fixed, the longer the application execution takes, the lower priority the application will have to use the resources. When the *variable* priority policy is applied, a suspended application does not age. This prevents an application from live-lock. Contrary to the suspending technique, the masking technique has no effect on application ageing as it does not suspend its execution.

The default policy is *variable*. However, it is possible to set the priority policy as an environmental constraint.

Application Priority: The priority of an application is the concatenation of two digits. The left digit represents the user priority and the right one represents the application priority. 10 categories from 0 to 9 have been considered to categorise users in the system. Each user can have different applications. 10 categories from 0 to 9 have been considered to categorise applications of a user in a system. Concatenating the two digits forms a number between 0-99 to represent an application priority. The priority of all components of an application is equal to the priority of the application.

Priority Observer: This method is invoked by the DSRI sever at any priority interval checkpoint. It adjusts the priorities of active applications if the *variable* policy is selected. Once

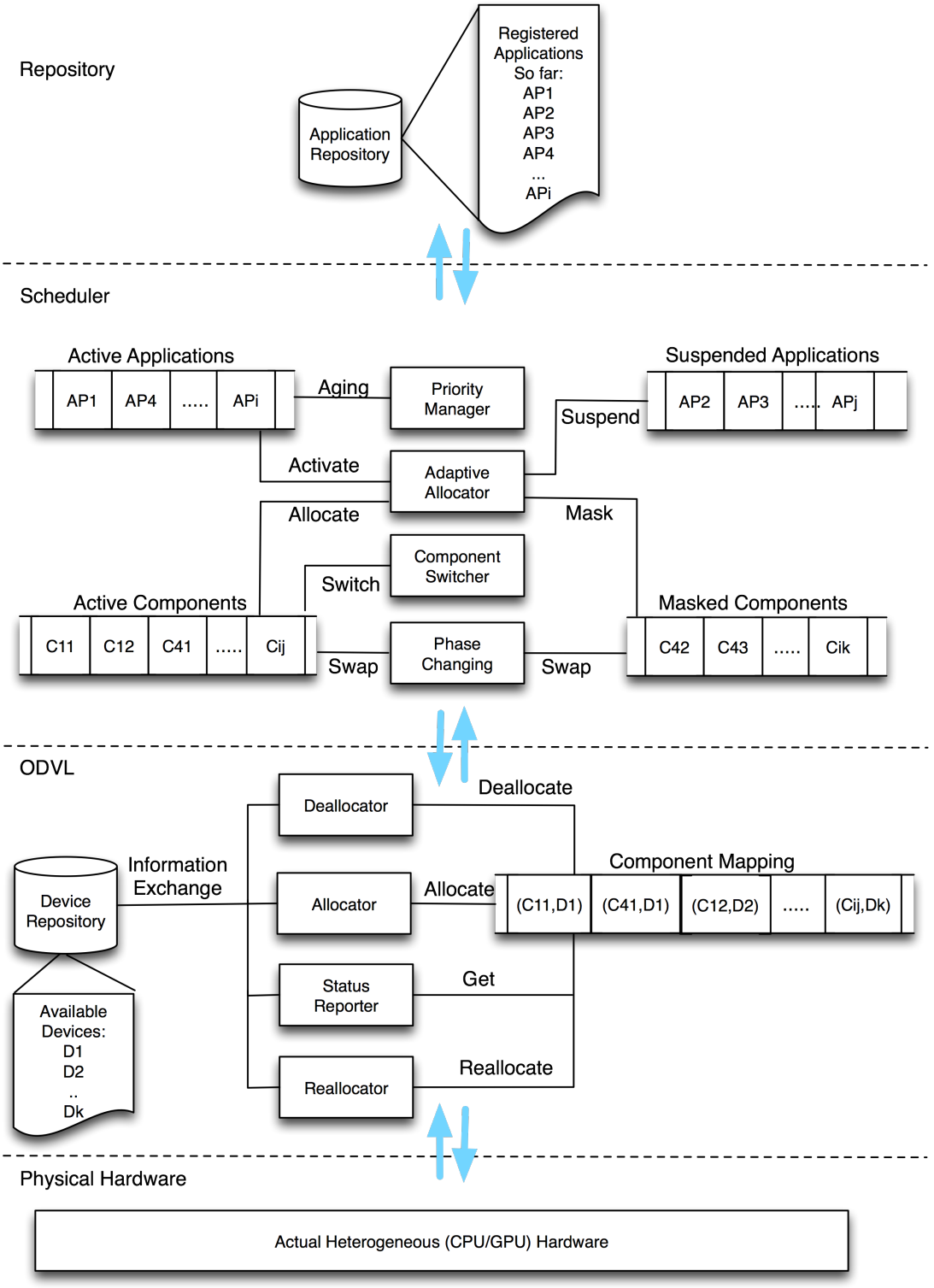


Figure 4.5: An architectural view of the Proposed OpenCL Scheduler and its relationship with ODVL and application registry. The blue arrows represents the communication mechanism. The solid lines connecting a module to a queue represent the type of operation applied by the module in the specified queue

adjusted, it tries to resume the suspended applications according to their priority. It is also invoked when an application terminates.

4.4.3.2 Adaptive OpenCL Allocator

When required, our adaptive OpenCL allocator dynamically reschedules a component on a new device. The new allocated resource will be used to execute the component for the next input iteration.

Selection Policy: Chooses a device with higher *computing power* for a component according to the priority of its application.

The *computing power* of device i for component j can be defined as follows:

$$CP_{ij} = \frac{cActiveTime_{ij}}{\sum_{k=1}^{N} cActiveTime_{kj}} \tag{4.10}$$

where $cActiveTime_{ij}$ is the active service time of component j on OpenCL device i; and N is the total number of available OpenCL devices. The default value for each device CP_{ij} is 1. A selected device can be either a *free device* or a *fully occupied device*. A *free device* is a device in which the number of allocated components is less than its determined `devCAP`. A *fully occupied device* is a device where the number of components running on it is equal to its provided `devCAP`. When a device is fully occupied, the following competition rule will be applied.

Competition Rule: Considering its priority, a component can compete for devices taken by other components with lower priorities.

Device Selection: Applying the selection policy, the device allocator runs a heuristic method to find an optimum device for a component. The following scenarios are possible when the device selection method is called.

- A *free device* is selected for a component. In this case the device will be allocated to the component and the device entry in the `deviceSet` will be updated.
- A *fully occupied device* is selected for a component. Then, a victim selection function is called to find a component with the lowest priority as a victim component. A FIFO policy is applied for components with the same priority. The victim component is replaced by the new component and the device entry in the `deviceSet` is updated. Then, the device selection method runs recursively to find another device for the victim component. The algorithm iterates until either no victim component is left or no device is found for the

victim component. When no device is found, the suspension technique is applied to the application containing the victim component.

Suspension Technique Depending on the structural pattern of an application, the two following techniques can be applied.

Application suspension: We define a suspension method to suspend an application whenever it has least one component for which no free device slots are available. It supports the atomicity rule where all resources allocated to the application will be released. This is a general technique that is applicable to all structural patterns.

Component Masking: Is applicable to the *parallel* building block ($[*]_N$). In HFastFlow, the *parallel* building block has been implemented as a farm pattern. When possible, our scheduler applies component masking to prevent an application from unnecessary suspending. Masking refers to temporarily or permanently suspending a farm worker without suspending an entire application. In this case a resource that is allocated to a masked worker is released without suspending the application. This would be useful if there are not enough resources available for an application execution. For a farm with N number of workers representing $[*]_N$ computation, it is possible to mask $N-1$ workers simultaneously.

4.4.3.3 Component Switcher

The component switcher method is defined to relocate the bottleneck node of an application. If a new device is detected for the bottleneck node, the ODVL reallocation method is invoked to reallocate the bottleneck component.

If the bottleneck node is an OpenCL component, the switcher method tries to dynamically select a device with higher *computing power* for the component.

When there are two devices with the same *computing power* the one with the lower *occupancy rate* will be selected. A device *occupancy rate* (OR) is calculated as follows:

$$OR = rn_d / mx_d \tag{4.11}$$

Where rn_d is the number of kernels currently running on the device and mx_d is the maximum number of kernels that can run per device.

4.4.3.4 Application Phase-Changing Detection

An application state can change during execution. Such changes can be a component termination or a bottleneck node variation. By detecting such changes during an application execution, it is possible to improve its performance. By knowing the structure of an application, it is

possible to detect such variations and reschedule the component allocations accordingly. Our OpenCL scheduler is equipped with two mechanisms called *resource stealing* and *resource sharing* to further refine scheduling decisions.

Resource stealing: Applies to components of a farm pattern for both $MISD$ and *parallel* computation. Inspired by a task stealing mechanism, a bottleneck OpenCL worker steals a resource with high computing power from the faster OpenCL worker upon its termination state.

In the task stealing approach, where an external queueing system and lock mechanism are applied, a faster worker steals the task from a slower worker once it has exhausted all its allocated tasks. However, the task stealing mechanism is not applicable to HFastFlow due to an internal queueing system that has a lock-free mechanism. In this case, each component has its own internal queue and does not have any access to other component queues.

Employing the structural information of an application, the proposed method detects farm patterns and the termination state of a faster worker. It then masks the worker and reallocates the bottleneck worker to the stolen device.

Resource sharing: Applies to a pipeline pattern where there is more than one OpenCL stage.
Monitoring the state of each stage, the proposed method:

- Finds the bottleneck stage and the device allocated to it as an old device;
- Finds a high *computing power* device for the bottleneck stage as a new device;
- Reallocates the bottleneck stage to the new device;
- Randomly selects one of the allocated stages to the new found device as a victim stage; and
- Reallocates the victim stage to the old device.

In this case the bottleneck stage in the pipeline will be periodically changed in order to help the flow of the stream in the pipeline. This process will be repeated periodically until the application is terminated.

At each monitoring time interval, a stochastic algorithm is applied to decide whether or not the resource sharing method should be used.

4.4.3.5 Scheduling Example

Figure 4.5 indicates a snapshot of the proposed OpenCL scheduling system that is mapping 4 different registered applications AP_1 to AP_4 (where the priority of $AP_1 > AP_4 > AP_2 > AP_3$)

on two different registered devices D_1 and D_2 (where $\texttt{devCAP}_{D1} = 2$ and $\texttt{devCAP}_{D2} = 1$). The applications AP_1 and AP_4 are running while the applications AP_2 and AP_3 are suspended. Also, application AP_4 has two pending components that are yet to be allocated in the masking queue while all components of AP_1 have already been allocated and running. The ODVL layer represents the mapping patterns of components to different available heterogeneous devices.

4.4.4 Static Structural Configuration

Tuning a structured application for a given architecture might produce an unpredicted result in another architecture. Therefore, fine tuning the application and mapping its components to the available resources can vary according to the given execution environment. As an external coordination method, the static structural configuration tries to automatically tune a structured application and to map its components to the available resources for a given environment in order to maximise the application throughput.

When no historical data is provided for an application, our static structural configuration requires a training period on the application in order to generate the sensor information such as the service time of the components.

By communicating with the DSRI server, the structural configuration receives an SKIP compliant structural configuration of the application and sensor information as an input. It generates an actuator file that contains the optimised SKIP compliant object which represents the application structural tree for the specific environment.

4.4.4.1 Abstract Computation Graph (ACG)

For an application, in order to optimise the number of instances for each building block component, we need to determine the data flow graph for an application. Therefore, depending on the data flow demand and the service time of a component we can optimise the number of instances for the component. Using the pattern based structured parallelism, it is possible to extract the data flow graph from the structural graph.

In this case, for any application constructed by the building block grammar represented in Listing 3.1, a new *abstract computation graph (ACG)*, representing the data flow of an application, can be generated by using the the reduction rules provided in equation 4.12.

$$
\begin{aligned}
&\wedge^{x} ::= \wedge \\
&\wedge\wedge ::= \wedge \\
&(\wedge C)^{x} ::= \wedge C^{x} \\
&(\wedge C\wedge)^{x} ::= \wedge C^{x}\wedge \\
&[C]_{x} ::= [C]_{x} \\
&[C_1,...C_x] ::= (C_1 \vee ... \vee C_x)
\end{aligned}
\tag{4.12}
$$

where

$\wedge ::= \dashv^{1x} \mid . \mid \vdash^{x1}$

represents *service component*s and

$C ::=$ «code» | ‹|code|› | «||code||»

represents *computing component*s. $\vee$ separates the choices in *MISD* computation.

Using the reduction rules in 4.12, the application structural tree will be converted into ACG where the computation components are separated from the service components. We refer to a service component as an *abstract queue*.

The throughput of an abstract queue is the ratio of the queue input rate to the output rate. The maximum value for queue throughput is 1, where the rate of the consuming tasks is equal to that of the producing tasks.

***Consuming Set*:** For an abstract queue, a *consuming set* refers to a set of computing components that consume from it. As the last abstract queue does not have a consuming set, the last $\wedge$ in ACG is eliminated.

***Component Configuration*:** For an abstract queue containing $[C_i]_x$, *component configuration* refers to the decision of determining x as the number of instances for member C_i of the queue consuming set. For an *HWRAPPER* node («||code||»), where the allocation of a component on different heterogeneous devices is possible, the decision to select the device is also considered as part of the configuration.

***Structural Configuration*:** For an application, *structural configuration* refers to determining the component configuration of all abstract queues.

The objective is to maximise the throughputs of all abstract queues and to keep the throughputs of all queues as close to each other as possible considering the resource limitation. In this case the number of resources allocated to each task queue is a key factor in achieving the objective.

Therefore, all possible structural configurations that satisfy the resource constraint will generate the solution space.

4.4.4.2 Monte-Carlo Tree Search (MCTS)

The Monte-Carlo Tree Search (MCTS) is a heuristic iterative optimisation algorithm, which is applied on the solution space to find an optimum solution. At each iteration, the MCTS selects a solution, simulates the execution of the problem for the selected solution and rewards it by analysing the simulation result based on our objective. The result of the reward will be back-propagated based on the applied back-propagation policy [92]. In the following we explain the proposed MCTS-based approach in more details [93].

Decision Tree: The MCTS approach operates on a decision tree created from the solution space. In a decision tree each depth corresponds to an abstract queue:

- A root of the tree represents 0 configuration.
- Nodes at depth 1 represent possible configurations for the first abstract queue in ACG.
- Nodes at depth 2 represent possible configurations for the second abstract queue in ACG.
- A path from a root to a leaf represents a candidate structural configuration as a solution for the specified environment.

For each queue the component configuration range is between 1 and the maximum available slots on all resources. The lower bound ensures that ACG will not be disjoint. The upper bound applies the resource limitation.

Each node represents a possible component configuration:

$$\{A(w_i,k) \mid w_i \in M, k \in \{0,1,\ldots,L\}\} \tag{4.13}$$

where $A \in \{\text{ADD}, \text{REMOVE}\}$; M is the set of components of the task-queue (that contain one element if there is only a CPU or a GPU component, or two elements if there is both a CPU and a GPU component); k is the amount of resources to be allocated to a component, w_i; and, L is the maximum amount of resources that can be allocated to a component.

Each path from the root to a leaf is considered as a viable solution to the problem.

Selection Strategy: The selection strategy applied here is *Upper Confidence bounds applied to Trees* (UCT) [94, 95]. The formula for UCT is:

$$UCT = \overline{X_j} + 2C_P \sqrt{\frac{2\ln n}{n_j}} \tag{4.14}$$

where n is the number of times the current node has been visited; n_j is the number of times the child, j, has been visited; $C_P > 0$ is a constant value; and, $\overline{X_j}$ is the average reward value given to child node, j.

Simulation of a selected solution: We have developed a simulator which mimics the behaviour of a given application running on the target architecture. The simulator outputs the metrics (queue level, resource utilisation, task-queue throughput) that we evaluate when tuning a given application.

The simulator builds a virtual environment via SKIP object environmental information.

For each node in ACG a component with a predefined service time is generated. For each task-queue in ACG a vector queue will be generated. Each component will be connected to an input queue and output queue which pops a task from input queue and pushes the result to an output queue.

There are two sets that represents the available CPUs and available GPUs and two queues that contain the active and suspended components in the system. Based on the component types (CPU or GPU), if an appropriate resource is available a component will be dedicated to it for a predefined time slot and it will move to the active queue. Otherwise the component will be put in the suspended queue.

To execute each task, a component may need several time slots of a resource. When a time slot ends, if the suspended queue is not empty, the component releases the resource and is moved from the active queue to the suspended queue and the resource will be given to another component in the suspended queue.

Each component receives a task from its input task queue and sends the result to its output task queue. When the input/output task queue is empty the component will release the resource and moves to the suspended queue.

The time unit can be scaled up in order to reduce the simulation time.

Reward Function Once a solution has been simulated, a reward for it is calculated. The reward function is based on the throughput of the system, denoted by T. There are two balancing factors related to the overall utilisation of the system:

1. We define the *utilisation factor* as SD_U, which is the standard deviation from the mean utilisation of all components in the system:

$$SD_U = \sqrt{\frac{\Sigma_i (CU_i - CU_{mean})^2}{N}} \tag{4.15}$$

where N is the total number of components in the system; CU_i is the utilisation of the component, C_i; and, CU_{mean} is the average utilisation of all components in the system. Using SD_U in a reward function keeps the number of instantiations for a component within a reasonable range. The lesser the effect of an instance of a component on application speed-up, the greater the SD_U value is. Also, this reduces unnecessarily usage of the resources.

2. We define the *throughput factor* as SD_Q. This is the standard deviation of the mean queue throughput, defined as follows.

$$SD_Q = \sqrt{\frac{\Sigma_i (T_{Q_i} - T_{mean})^2}{L}} \tag{4.16}$$

Here, L is the total number of component queues in ACG, T_{Q_i} is the throughput of component queue Q_i and T_{mean} is the average value of the throughput of all component queues in ACG. Adjusting the reward for this factor discourages the allocation of additional resources to the components of a component queue when they are no longer bottlenecks on that component queue.

The reward function for a selected path, v, of the decision tree is calculated as follows:

$$Q(v) = T - (SD_U + SD_Q) \tag{4.17}$$

Back-propagation, Termination Condition and Final Move Selection: We have considered the *average* back-propagation policy, where the average reward of all children is propagated to the parent [96].

The MCTS algorithm finishes if no new moves are made for K iterations. The final move selection is based on the *robust-max child* policy. To select the final path in each step, the robust-max child policy tries to select the child with both the highest visit count and the highest value. If there is no robust-max child at any step, more simulations are run until a robust-max child is obtained [96].

4.5 Summary

In this chapter we have implemented a SKIP Compliant autonomic behavioural system, called PEI that is capable of coordinating a structured application both static and dynamically.

The instrumented framework, coordination engines and interaction mechanism are physically located in different packages where each package can be considered as an independent process interacting via the SKIP methodology with each other. Applying UML notation, Figure 4.6 demonstrates a component diagram representing the physical view of PEI system.

AS stated in figure 4.6, each application has its own DSRI client located inside the application executing in a separate thread. During application execution, the DSRI client invokes the application getter method from the root component in an application tree structure. Through the getter method the SKIP compliant extra-functional information is recursively extracted from all components of the application in the form of a sensor file. Once extracted, the DSRI client sends the sensor information to the DSRI server via ZMQ proxy. For each application,

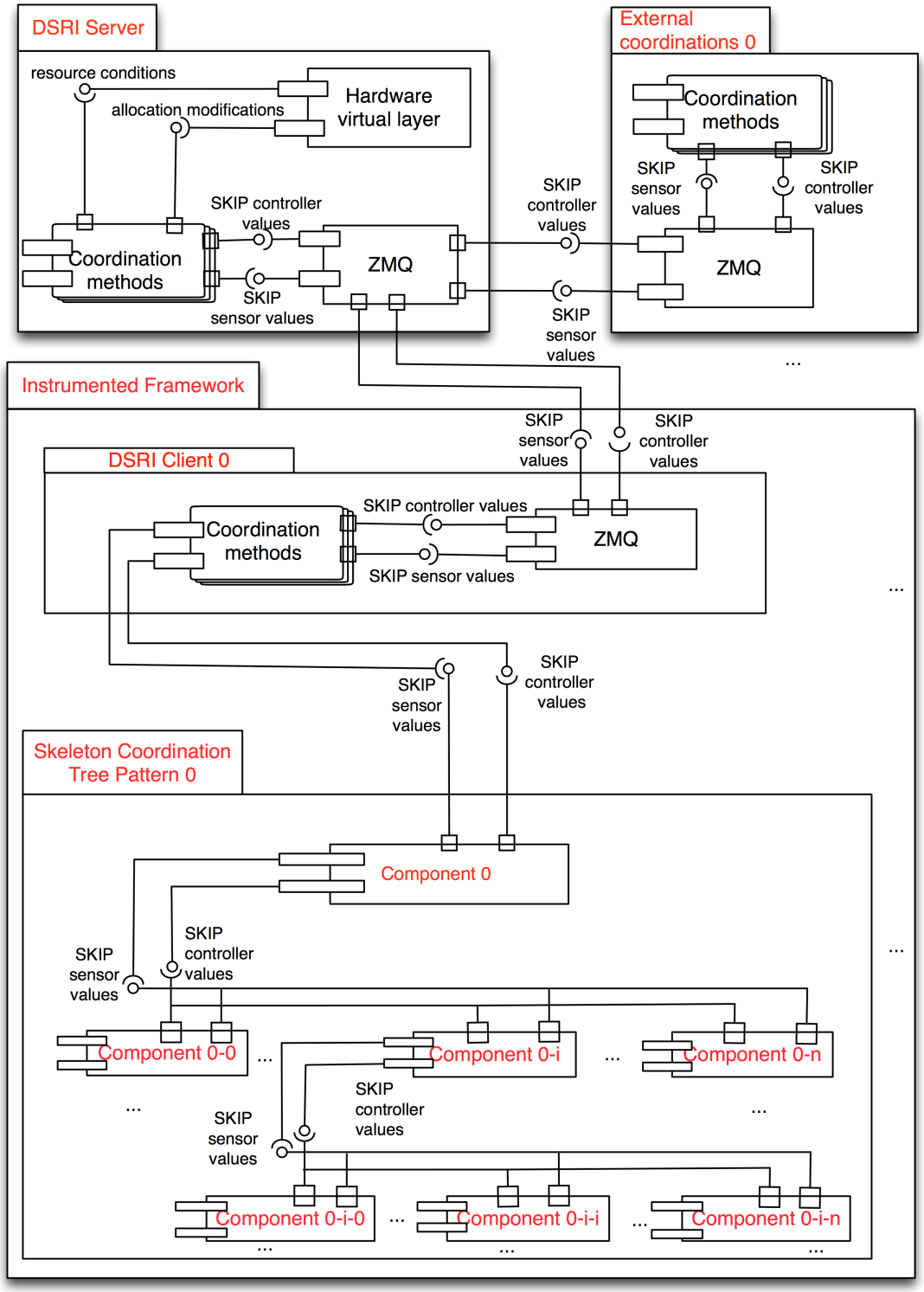


Figure 4.6: A component diagram representing the physical view of the autonomous behavioural framework

the DSRI server, which is located in a separate process, receives the sensor information; dispatches it through both internal and external coordination engines via ZMQ proxy; and receives the coordination modification instructions in the form of actuators. The actuators are sent to the applications by the DSRI server. Once an actuator file is received, the DSRI client invokes the application setter method from the root component in an application tree structure. Through the setter methods, the actuator file is recursively distributed among all the components of the application from root to leaves and the appropriate coordination modifications are applied to each component.

Using the HFastFlow framework which implements the skeleton patterns through RISC-pb^{2}l approach, this provides a certain level of abstraction in communication and synchronisations over heterogeneous multi-core applications.

Extending the HFastFlow framework by implementing the OpenCL back-end demonstrates the adaptability and expandability of the RISC-pb^{2}l approach for targeting heterogeneous multi-core architectures.

We have demonstrated the different actuators provided to enhance the HFastFlow controllability. Setting the actuators' parameters can be autonomic and manual. Typically, actuators contributing to the application structure such as queue size, OpenCL component tuning, and efficient idling can be set statically before application execution. Such decisions depend on the structure of the application and the chosen environment. However, actuators such as component allocations and workload distributions can vary during an application execution. Static configuration of such actuators may not improve an application's performance due to the non-deterministic nature of the dynamic changes in resources and the application execution conditions. Decisions about such actuators vary depending on the observed conditions of both the application and the environment. Each decision can be different from the previous one.

SKIP provides a mechanism for exchanging the information between the skeleton framework and the coordination methods. With a platform independent interaction protocol, SKIP provides a means of communicating between the coordinating engines and the executing applications. We use a 2-level hierarchical DSRI system with a certain level of autonomy for the subordinate. While the client is responsible for executing the application and extracting/injecting the SKIP compliant objects, the server has the responsibility of coordinating the resources among all running applications.

Using the client-server architecture for DSRI isolates the overhead of the coordination decision from user applications. By applying the hierarchical technique, each skeleton application is responsible for controlling its coordination workspace with a certain level of autonomy in the provided resources, while the allocation of the resources to each application is handled by the server-side coordination engines.

Also, applying a non-blocking asynchronous message passing technique between the client

and server prevents the client from blocking the coordination decisions while exchanging the information. Moreover, determining the DSRI client as an independent thread and using a single mutual exclusion technique for shared parameters separates the cost of communicating with coordination decisions from user applications. With global knowledge of all executing applications, the DSRI provides multi-tenant application execution over the heterogeneous multi-core system.

The provided OVDL is used to unify the underlying heterogeneous OpenCL devices and to separate the resource management from the executing application.

Connecting the instrumented HFastFlow framework and providing coordination engines through DSRI, it is possible to provide an autonomic and semi-autonomic controls over applications dynamically with the assistance of SKIP to exchange information. While autonomic controls are provided by coordination engines for actuators that require dynamic decisions, with the assistance of a user, setting the structural actuators required for static decisions yields semi-autonomic controls over an application.

As stated in Appendix A, although the composition rules provided in the SKIP methodology is capable of generating and instrumenting all parallel patterns that is supported by RISC-pb^{2}l, the operability of these patterns depends on their implementation through the HFastFlow framework. Currently HFastFlow only implements the RISC-pb^{2}l compositions that is required for farm and pipeline patterns. Therefore, full operability of all the compositional features of RISC-pb^{2}l libraries will be available when these compositions are implemented by the HFastFlow framework. In the next two chapters we evaluate the applicability of the SKIP methodology for different categories of applications implemented through HFastFlow skeleton-based parallel patterns through RISC-pb^{2}l libraries using PEI.

Chapter 5

Evaluation of OpenCL Based Applications

In this chapter we evaluate the performance overhead of PEI and coordination optimisation achieved by PETs for PEI-based applications. All applications in this chapter use the HFast-Flow implementation of *HWrapper*, the OpenCL back-end presented in Chapter 4, section 4.1.1.

By using *HWrapper*, we can differentiate between CPU based and GPU based components and extract the specific extra-functional information for GPU components. This information can be used for tuning the coordination of a structured application on a heterogeneous (CPU/GPU) multi-core architecture as stated in Chapter 3, section 3.2.

This chapter is composed of three sections. In section 5.1 we present the application suite that has been developed to evaluate our SKIP compliant framework. In section 5.2, we thoroughly evaluate our approach for the designed test suite over different platforms. Finally, a summary of this chapter is provided in section 5.3.

5.1 Application Suite

We emphasise that our intention here is to demonstrate the benefit of our *HWrapper* in further control and tuning the coordination mechanism. Therefore, we do not intent to develop a new version of the tuned OpenCL kernel for each application. It this spirit, in order to develop the following application, we have applied the OpenCL kernel provided by AMD [97] for all the implemented benchmark applications.

Due to their SIMD parallelism, the performance of GPU implementations greatly depends on how easy it is to make a parallel adaptation of a given algorithm. Image processing algorithms are often massively parallel by nature, since the parallelisation is naturally provided by per-pixel (or per-voxel) operations. This simplifies their implementation in a GPU and in general make them good candidates for GPU implementations [98]. Therefore, image processing

applications have been considered for developing the test suite that executes over heterogeneous architectures.

We have designed six different applications to cover the supported parallel patterns by HFastFlow. The provided applications aim to cover cases with *i)* different types of parallel patterns in HFastFlow (Sobel Filter, URNG, and Gaussian Noise); *ii)* nested composition of supporting patterns (Bilateral Denoise and Separable Convolution); and, *iii)* more than one OpenCL kernel per application (Recursive Gaussian).

Table 5.1 represents the characteristics of each application. The first column of the table represents the application name and the second column represents the features that the application covers. The applications aims to cover different coordination features for different RISC-Pb^{2}l parallel patterns supported by HFastFlow.

Table 5.1: Summary of Applications Characteristics.

Application Name	Application Characteristics
Sobel Filter	Represents a heterogeneous Pipeline pattern in HFastFlow with an OpenCL component in the second stage. **Not** generated by SKIP adapter, this application demonstrates the applicability of using coordination engines for existing applications using *HWrapper*.
URNG	Represents a heterogeneous Pipeline pattern in HFastFlow with an OpenCL component in the second stage. This is an example of Heterogeneous Pipeline pattern that is generated by SKIP adaptor.
Gaussian Noise	Represents a heterogeneous Farm pattern in HFastFlow with OpenCL components as Farm workers. This is used to demonstrate the coordination of different OpenCL workers in a Farm pattern.
Bilateral Denoise	Represents a nested composition of heterogeneous Farm and Pipeline Patterns in HFastFlow with OpenCL components as Farm workers. The Farm pattern is the second stage of the Pipeline and covers the reduction composition in RISC-pb^{2}l (i.e. the Farm pattern **with** collector in HFastFlow).
Recursive Gaussian	Represents a heterogeneous Pipeline pattern in HFastFlow with **more than one** OpenCL components. This is used to demonstrate the coordination of **more than one** OpenCL components in a Pipeline pattern.
Simple-Convolution	Represents a nested composition of heterogeneous Farm and Pipeline Patterns in HFastFlow with OpenCL components as Farm workers. The Farm pattern is the last stage of the Pipeline and covers the non-reduction composition in RISC-pb^{2}l (i.e. the Farm pattern **without** collector in HFastFlow).

The SKIP adaptor has been applied to develop the following applications in HFastFlow. Using the benefit of PEI, *(1)* the application structure; *(2)* the applied pattern for each component; *(3)* the number of workers in farms; *(4)* queue bound; *(5)* the type of load-balancer, *(6)* an application priority; and *(7)* the load-balancer policy are tunable parameters which can be changed for each instance of application. Applying our SKIP adaptor also allows the pro-

grammer to have different configurations of the above parameters for an application without the need to recompile the code.

In order to visualise the structure of an application, we demonstrate the structural composition of each application in a tree format referred to as its *structural tree notation*. Therefore, we have generated a notation for each HFastFlow component. Table 5.2 represents the correspondence between the structural tree notations and the HFastFlow components.

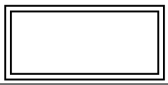
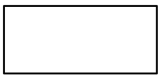
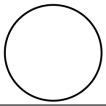
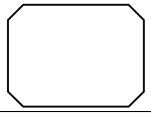
Structural Tree Notation	HFastFlow Component
	OpenCL wrapper `OclNode`.
	CPU wrapper `ff_node` and filter function wrapper encapsulated in `ff_loadbalancer` and `ff_gatherer`.
	Pattern container `ff_pipeline` and `ff_farm`.
	Combinators `ff_loadbalancer` and `ff_gatherer`.
	Parallel execution of a set of workers for `ff_farm`.
	It represents a non-FastFlow component executing unit of computing function.
	Representing `feedback` for a HFastFlow pattern container.
	It represents that component *A contains* component *B*.
	Controlling function `barrier` can be incorporated in a HFastFlow component .

Table 5.2: The correspondence between structural tree notations and HFastFlow components

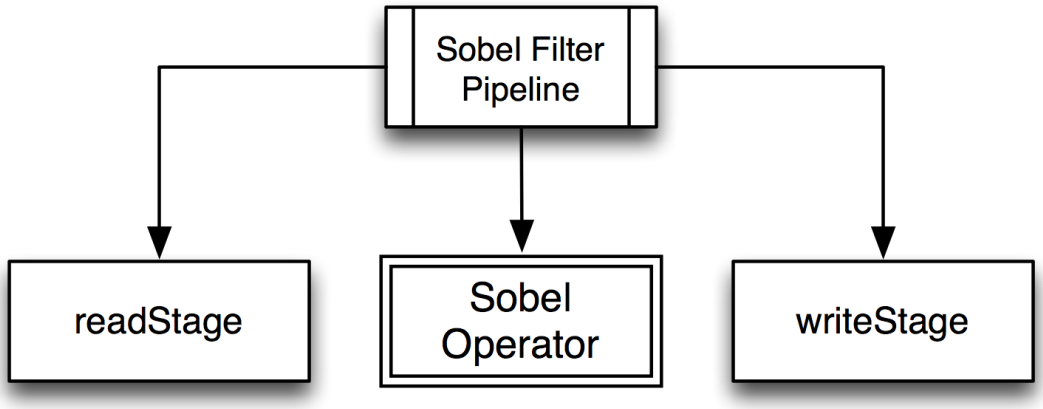


Figure 5.1: The structural composition of components for the Sobel Filter application in HFastFlow visualised by structural tree notation

5.1.1 Sobel Filter

The sobel operator [99] creates an image which emphasises edges and transitions. The input buffer is stored in a 2D image buffer to cache neighbouring pixel reads. Each work-item calculates the Dx and Dy of a pixel by applying 3x3 filters on nine pixels, including the pixel itself. The final pixel value is written by calculating the gradient magnitude of the partial derivatives. The application is composed of three parts: `readStage`, `sobelOperator`, and `writeStage`.

Using RISC-pb^{2}l building block notations we have designed the application as follows:

«readStage » **.** «||sobelOperator||» **.** «writeStage »

Appendix B.5.1 represents the generation of the sobel filter through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.1 represents the architectural view of the application in HFastFlow. Using the SKIP-compliant object represented in Appendix B.5.2, our adaptor for HFastFlow translates it into a three stage pipeline where the first stage reads the stream of images from different files, the second stage applies the sobel operator to the image and the third stage stores the image on the provided output folder.

5.1.2 Bilateral Denoise

A bilateral filter [100] is a non-linear, edge-preserving and noise-reducing smoothing filter for images. The intensity value at each pixel in an image is replaced by a weighted average of intensity values from nearby pixels. This weight is based on considering both the spatial distance and the colour distance between its neighbours. This preserves sharp edges by systematically looping through each pixel and adjusting weights to the adjacent pixels accordingly. The application is composed of three parts: `readStage`, `bilateralDenoise` and `writeStage`.

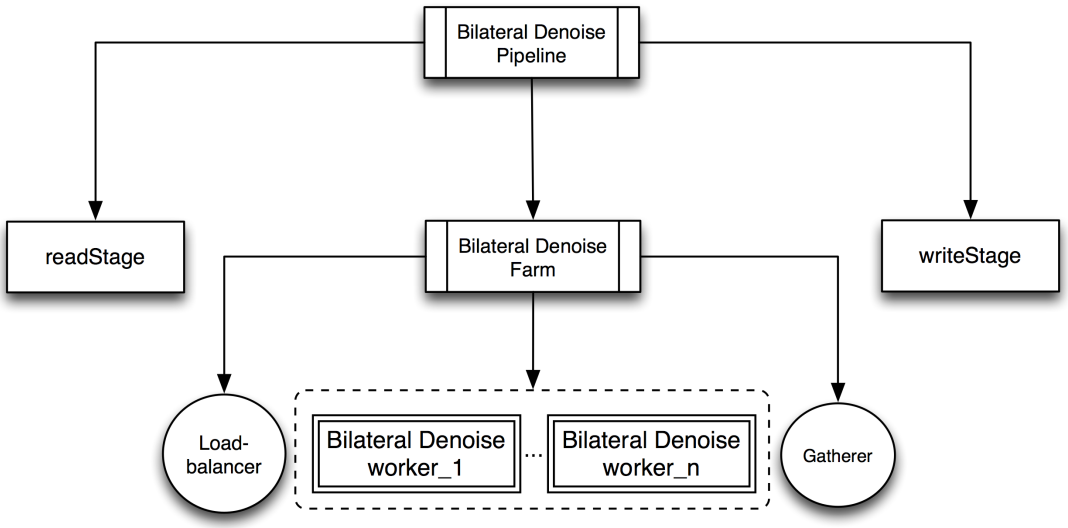


Figure 5.2: The structural composition of components for the Bilateral Denoise application in HFastFlow visualised by structural tree notation

Using RISC-pb^{2}l building block notations, we have designed the application as follows:

«readStage » . $Unicast_{1 \lhd n}$. $[\text{«}\|\text{bilateralDenoise}\|\text{»}]_n$. $gather_{n \rhd 1}$. «writeStage »

Appendix B.4.1 represents the generation of bilateral denoise through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.2 represents the architectural view of the application in HFastFlow. Using the SKIP-compliant object represented in Appendix B.4.2, our adaptor for HFastFlow translates it into a three stage pipeline where the first stage reads the stream of images from different files. The second stage is translated to a farm pattern that applies the bilateral denoise on the image. An instance of `ff_loadbalancer` with a unicast function is used as a filter to unicast images among the workers. Each OpenCL worker receives a task and executes the bilateral denoise filter on it. An instance of `ff_gatherer` with a gather filter is applied. It receives tasks from all workers and passes them to the third stage. The third stage stores the image on the provided output folder.

5.1.3 Gaussian Noise

Gaussian noise [101] is a statistical noise that has a probability density function of the normal distribution. The values that the noise can take on are Gaussian distributed. The application takes an input image and generates a Gaussian deviation by using the pixel value as a seed. This deviation is then added to all the components of the pixel.

The application is composed of three parts: `readStage`, `GaussianNoise`, and `writeStage`. Considering the spread and reduce functions that are capable of fusing custom computing functions with combinators, we design the Gaussian noise application as a single reduction compo-

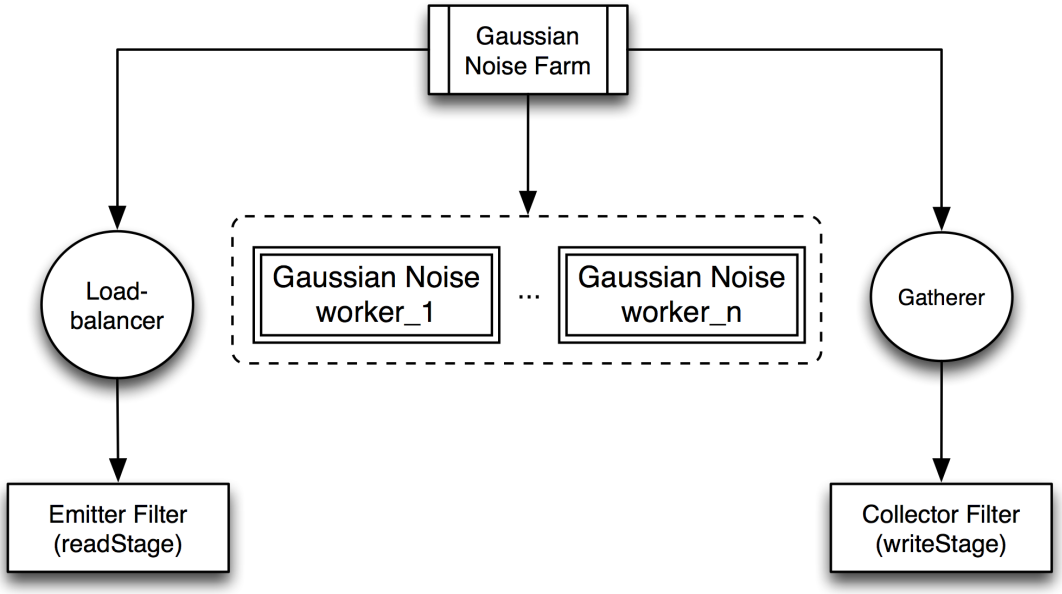


Figure 5.3: The structural composition of components for the Gaussian Noise application in HFastFlow visualised by structural tree notation

sition.

Using RISC-pb^{2}l building block notations, we have designed the application as follows:

$(ReadUnicast_{1 \lhd n})$. $[\text{«||gausianNoise||»}]_n$. $(Writegather_{n \rhd 1})$

Appendix B.6.1 represents the generation of Gaussian noise through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.3 represents the architectural view of the application in HFastFlow. Using the SKIP-compliant object represented in Appendix B.6.2, our adaptor for HFastFlow translates it into a farm pattern where the emitter is a custom spread function reading the stream of images from different files. Also, an instance of `ff_loadbalancer` with a unicast function is used to unicast images among the workers.

Each OpenCL worker receives a task and executes the Gaussian noise on it.

The gather function is also a custom function where an instance of `ff_gatherer` with a gather filter is applied to receive the tasks from all workers and stores the image in the provided output folder.

5.1.4 Uniform Random Noise Generator (URNG)

The URNG application generates noise in an image by using a linear congruential generator. It generates a uniform deviation in the range $(0, 1)$ and multiplies by a noise factor to produce the final noise. It calculates the uniform deviation from the seed, which is generated by averaging four components of a pixel. It then applies the deviation (multiplied by the noise factor) to

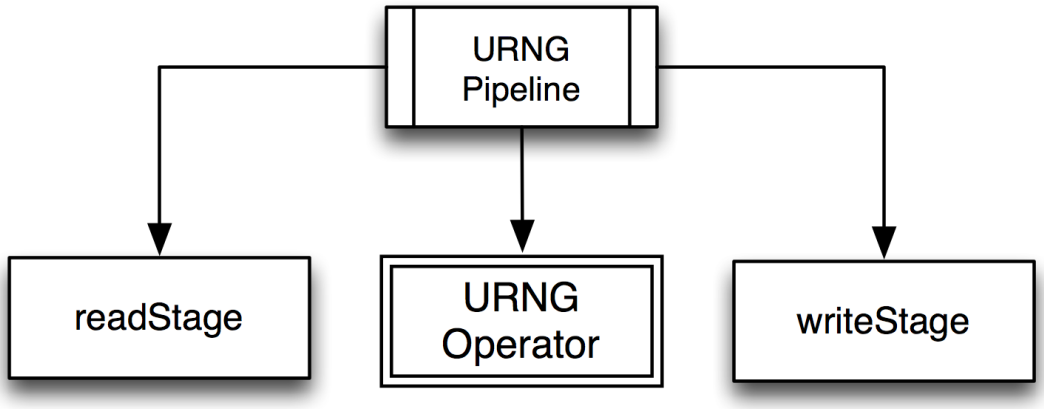


Figure 5.4: The structural composition of components for the URNG application in HFastFlow visualised by structural tree notation

all the components of the pixel. Thus, each global thread computes a uniform deviation and applies it to a pixel.

The application is composed of three parts: `readStage`, `URNG` and `writeStage`.

Using RISC-pb^{2}l building block notations we have designed the application as follows:

«readStage » **.** «||urng||» **.** «writeStage »

Appendix B.1 represents the generation of URNG through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.4 represents the architectural view of the application in HFastFlow. Using the SKIP-compliant object represented in Appendix B.1.1, our adaptor for HFastFlow translates it into a three stage pipeline pattern where the first stage reads the stream of images from different files and the second stage applies the URNG function on the image. The third stage stores the image in the provided output folder.

5.1.5 Recursive Gaussian

Since computational efficiency is often important, low-order recursive filters are often used for scale-space smoothing. The recursive Gaussian application implements a Gaussian blur using Deriche's recursive method [102]. The advantage of this method is that the execution time is independent from the filter width.

The design application is a pipeline with the seven following steps:

- Read input image.
- All rows of a given bitmap image matrix are processed separately with the method described in [102]. This basically consists of a forward run followed by a backward run through the row.

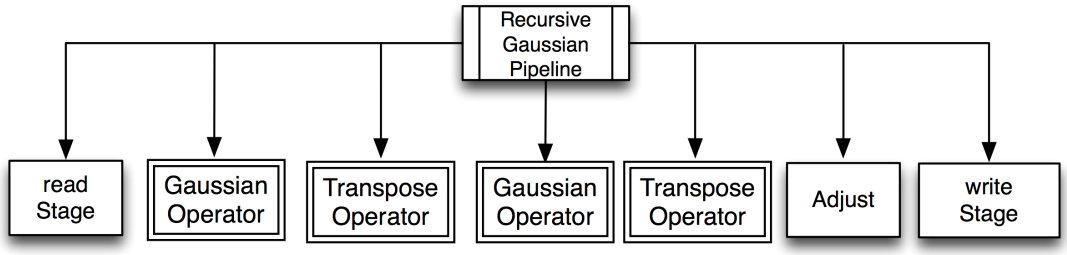


Figure 5.5: The structural composition of components for the Recursive Gaussian application in HFastFlow visualised by structural tree notation

- The bitmap image is transposed. Transposing a matrix means the first row becomes the first column, the second row becomes the second column, and so on.
- The second step is applied over the transposed bitmap image matrix.
- The third step is applied to rotate the bitmap image to its original form.
- Adjust the image direction for storing.
- Store the image.

Using RISC-pb^{2}l building block notations we have designed the application as follows:

«readStage »`.`«||gausianFilter||»`.`«||transpose||»`.`«||gausianFilter||»`.`«||transpose||»`.`«adjust »`.`«writeStage »

Appendix B.2 represents the generation of Recursive Gaussian through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.5 represents the architectural view of the application in HFastFlow. Using SKIP-compliant object represented in appendix B.2.2, our adaptor for HFastFlow translates it into a seven stage pipeline. Stages 2, 3, 4 and 5 are OpenCL stages that use the provided direct OpenCL back-end.

5.1.6 Separable Convolution

Image convolution is widely used in image processing applications, e.g. for blurring, smoothing and edge detection. Our version of image convolution consists of reading a stream of images into the memory and applying the `convolution` function (with a given filter) to each of these images. Applying the convolution function to an input image consists of calculating, for each pixel of the input image, a scalar product of the "window" that surrounds that pixel with the filter weights, and storing the result in the output image in the same position:

$$out(i,j) = \sum_m \sum_n in(i-n, j-m) \times filter(n,m),$$

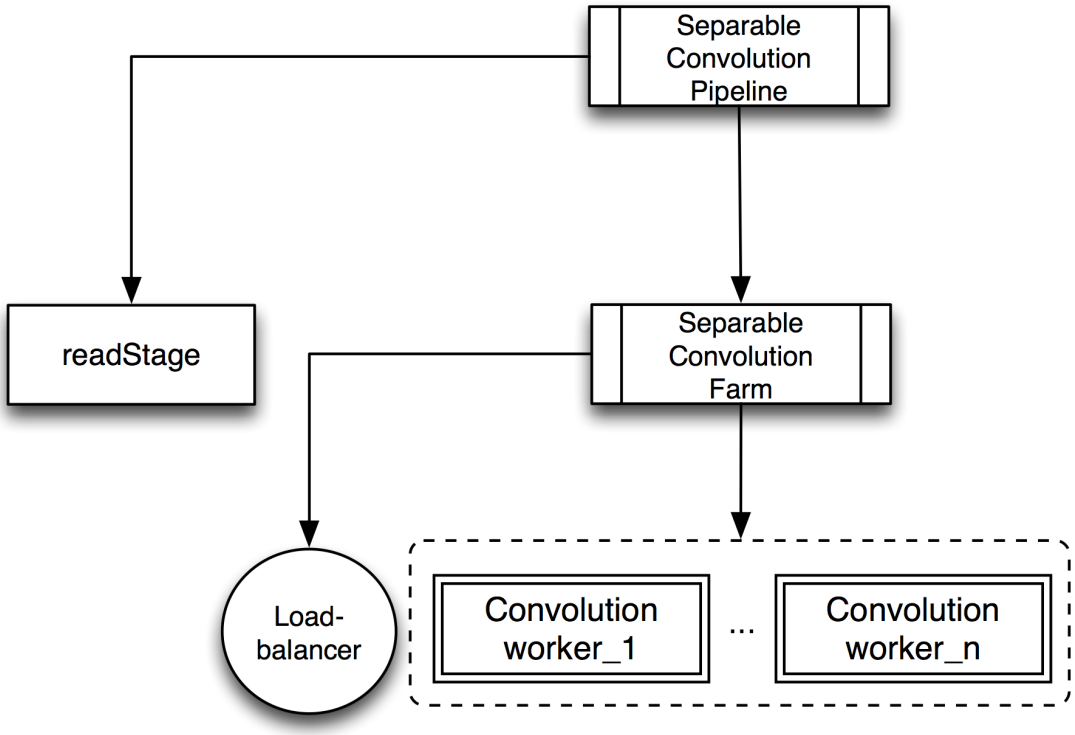


Figure 5.6: The structural composition of components for the Simple Convolution application in HFastFlow visualised by structural tree notation

where $out(i, j)$ is the pixel of the output image at position (i, j), $in(i, j)$ is the pixel of the input image at position (i, j), m and n are the dimensions of the filter, and $filter(i, j)$ is the pixel of the filter at position (i, j).

Here, the applied filter is a two dimensional matrix which can be expressed as the outer product of two vectors. Therefore, the application is called separable 2D-convolution.

The provided application is composed of the two following parts:

- **Generation** of suitable input matrix `input_pixel` and filter weights matrix `filter_weight`.
- **Convolution** to apply the `filter_weight` on `input_pixel`.

Using RISC-pb^{2}l building block notations, we have designed the application as follows:
«generation » . $Unicast_{1 \lhd n}$. $[$«‖convolutionFilter‖»$]_n$

Appendix B.3.1 represents the generation of the separable convolution through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 5.6 represents the architectural view of the application in HFastFlow.

Using the SKIP-compliant object represented in Appendix B.3.2, our adaptor for HFastFlow translates it into a three stage pipeline. The first stage generates `input_pixel` and

Table 5.3: Hardware Specification Table for the Titanic machine and the Xookik cluster

Parameter	Titanic	Xookik
CPU Model	AMD Opteron™ Processor 6176	Intel®Xeon®
No. of CPUs	2	2
Cores per CPU	12	6
CPU Clock (GHz)	2.3	3.07
physical Memory (GB)	32	50
No. of GPUs	1	1
GPU Model	NVIDIA Tesla C2050	NVIDIA Tesla M2090
GPU Memory (GB)	2.68	6
GPU Cores	448	512
CUDA Version	4.0 V0.2.1221	4.0 V0.2.1221
GPU Driver Version	290.10	290.10
OpenCL driver	AMD-SDK version 2.8	AMD-SDK version 2.8

`filter_weight`. It allocates a sequence of number as a unique ID for every input matrix.

The second stage is translated into a farm pattern that applies the convolution function to the generated matrices.

An instance of `ff_loadbalancer` with a unicast function is used as a filter to distribute the generated matrix among the workers.

Each OpenCL worker receives a task and executes the separable convolution on it.

5.2 Evaluation

An evaluation of PEI was carried out on two hardware platforms. Table 5.3 provides the detailed specifications of the applied test machines. The Xookik cluster has 4 symmetric nodes. Each node has the same specification as that provided in 5.3. The cluster is located at the Robert Gordon University, Aberdeen. The Titanic machine is located at the University of Pisa, Italy.

The execution time demonstrated for each application is the average runtime of the application over 20 executions.

5.2.1 Performance Overhead of PEI

As mentioned in Chapter 4, the DSRI client and server are in a separate process from each other. For each application, all interactions such as exchanging information with the DSRI server and external coordination, are performed through the DSRI client.

Although the DSRI client is in the same process as the skeleton-based application, it will be executed by a separate thread. This will provide a clean separation between the overhead of threads executing the application and that running the DSRI client.

Therefore, the overhead of PEI for executing an application is limited to the interaction between the DSRI client and the application through the `getter` and the `setter` methods.

For any intra process race conditions between the DSRI client and the components, such as reading or modifying the actuator and sensor parameters, a single mutual exclusion is applied. This means the components always win the competition to access the actuator and sensor parameters.

As the `getter` and the `setter` methods are invoked by the DSRI client, using the single mutual exclusion pushes the overhead of these interactions on the DSRI client rather than the application.

However, each application component is responsible for extracting its specified sensor information. Therefore, the granularity of the extracted information and the monitoring sensitivity are the two factors that determine the overhead of PEI for executing an application.

The monitoring sensitivity can be determined through a constraint object which can vary depending on the application nature. To determine the granularity of the collected sensor information, there are two sampling modes, `Sparse` and `Aggressive`.

In order to determine the framework overhead, we have compared the execution time of applications in the instrumented HFastFlow using sensors for each sampling mode with the execution time of the same applications in the non-instrumented version of HFastFlow. In order to avoid biased results, the controllers, coordination engines and dynamic adjustment are not applied here and all cases are executing in a same coordination configuration. Therefore, the execution time of the instrumented version is equal to the time spent on running the original HFastFlow plus the time spent on extracting the sensor data.

Moreover, in order to determine an example of runtime generated sensor files, a single snapshot of the extracted information for each application is presented in Appendix C.1.

Also, Appendix C.2 demonstrates a single snapshot of the actuator files generated by coordination engines for each application

Moreover, appendix C.3 represents a sample constraint file that is applied to all applications except recursive Gaussian.

Figure 5.7 demonstrates the performance overhead of the framework for extracting the extra-functional requirements for the different image processing applications developed here.

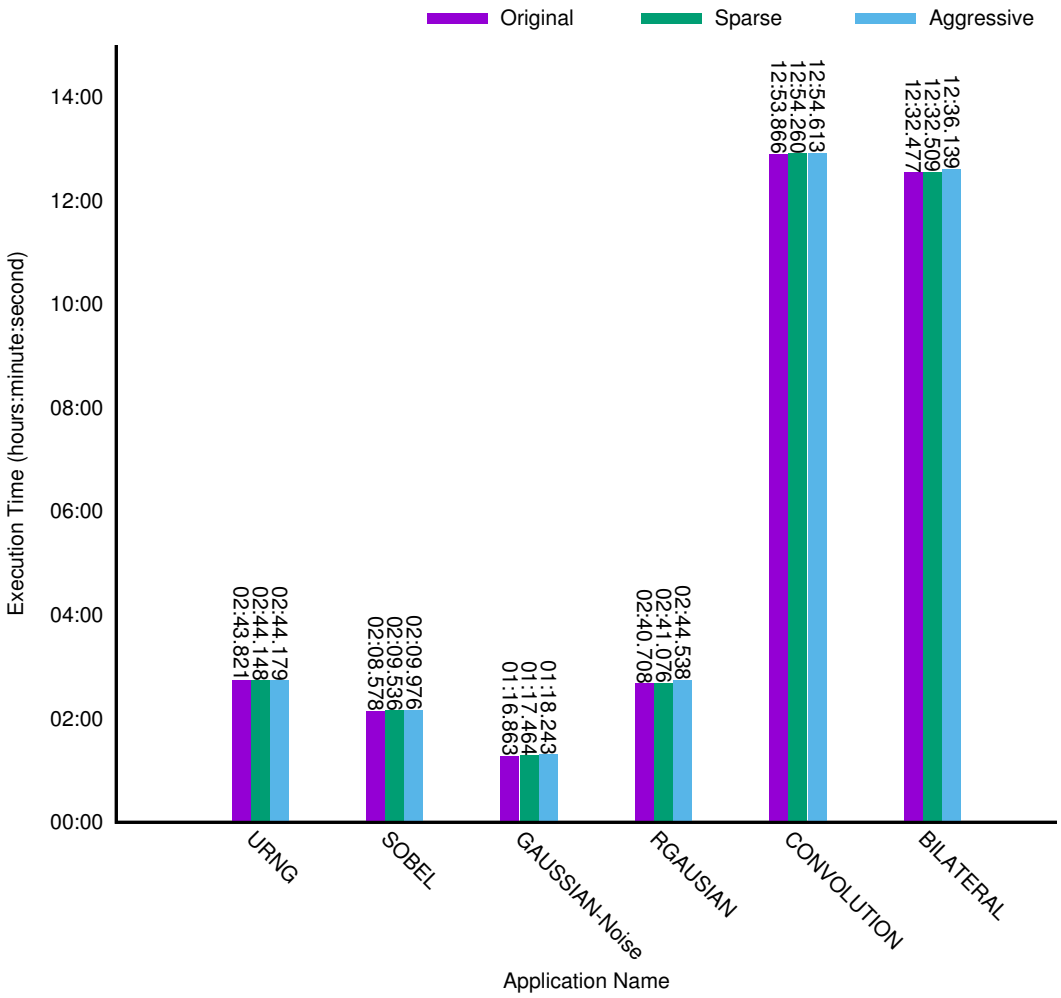


Figure 5.7: The upper-bound overhead of performance metrics tracing over the HFastFlow framework on a node of the Xookik cluster for image processing applications.

Table 5.4: Input stream specification for image processing applications

Image Type	Image Size	Input Stream Number
Bitmap	64*64	1024
Bitmap	128*128	1024
Bitmap	256*256	1024
Bitmap	512*512	1024
Bitmap	1024*1024	1024
Bitmap	2048*2048	1024

Each application receives the streams of 6144 images of 6 different size categories presented in Table 5.4. The original bar represents the execution time of an application without using the SKIP methodology; The Sparse bar represents the execution time of an application collecting sensor information in the sparse mode; and the aggressive bar represents the execution time of an application when collecting sensor information in the aggressive mode

As observed in Figure 5.7, the maximum overhead is for the aggressive version and is related to the recursive Gaussian application: 2.5%. For most cases this value is around 1%.

This evaluation also determines a performance lower bound for every coordinating method applied in the provided applications. Therefore, for PEI, a coordinating optimisation method that demands an aggressive information collection is acceptable if it optimises an application throughput of more than 3%.

It is obvious that in sparse mode information collection, which is the default mode for PEI, a minimum of a 1% performance improvement is required.

5.2.2 OpenCL Back-end

In this section we evaluate the HFastFlow OpenCL back-end, presented in Chapter 4, section 4.1.1, that uses the proposed PETs in PEI to tune coordination over heterogeneous multi-core architectures.

Table 5.5 states the the execution time of the sobel filter application on the Titanic machine. The application receives the streams of 6144 images of 6 different size categories presented in Table 5.4.

Parallelising the sobel filter application with HFastFlow, we can increase the GPU utilisation by 38%. Consequently, up to 1.63 times speed-up has been achieved over the serial version executing on a GPU device (the AMD version).

Application Name	GPU Utilisation	total Runtime (hh:mm:ss.milliseconds)
SOBEL-FF	95%	00:02:48.640
SOBEL-AMD	57%	00.04:35.913

Table 5.5: The execution times and GPU utilisation for the Sobel Filter application on the Titanic machine

Figures 5.8 and 5.9 present the execution of bilateral denoise and Gaussian noise respectively on a node of the Xookik cluster. Both applications receive the streams of 6144 images of 6 different size categories presented in Table 5.4.

The first column represents the serial execution of the OpenCL kernel on the multi-core CPU; the second represents the serial execution of the OpenCL kernel on the GPU; and the third represents the parallel execution of the application on HFastFlow. In both applications for the HFastFlow version, the second stage is a farm with two workers that use CPU and GPU devices. Also, an adaptive load-balancer for each worker has been applied to distribute the tasks among the workers in proportion with the service time of the workers. As stated in both figures, executing the application on a GPU device can achieve up to 9 times speed-up over that of a CPU device. Moreover, parallelising an application using a HFastFlow skeleton pattern represents 1.21 times speed-up over the serial version executing on a GPU (the AMD version).

Although the serial version of applications executing on a GPU device and those developed in HFastFlow use the same OpenCL kernel, the HFastFlow versions state improvements in both utilisation and execution time. The potential reasons of such improvement can be described as follows:

- Using the provided parallel skeleton pattern in HFastFlow, reading input, processing it and writing the output are considered as independent stages of a pipeline pattern. Applying the pipeline pattern, all components are executed simultaneously. This will increase the resource utilisation and reduces the application service time. Also, reading the next input stream while the OpenCL component is processing the previous one in the GPU increases the GPU utilisation, as it eliminates the waiting process of the GPU-allocated component for a task (5.5).

- For the bilateral denoise and Gaussian noise application, the CPU-allocated OpenCL worker assists the GPU-allocated one. This increases the resource utilisation and improves the service time of the farm component.

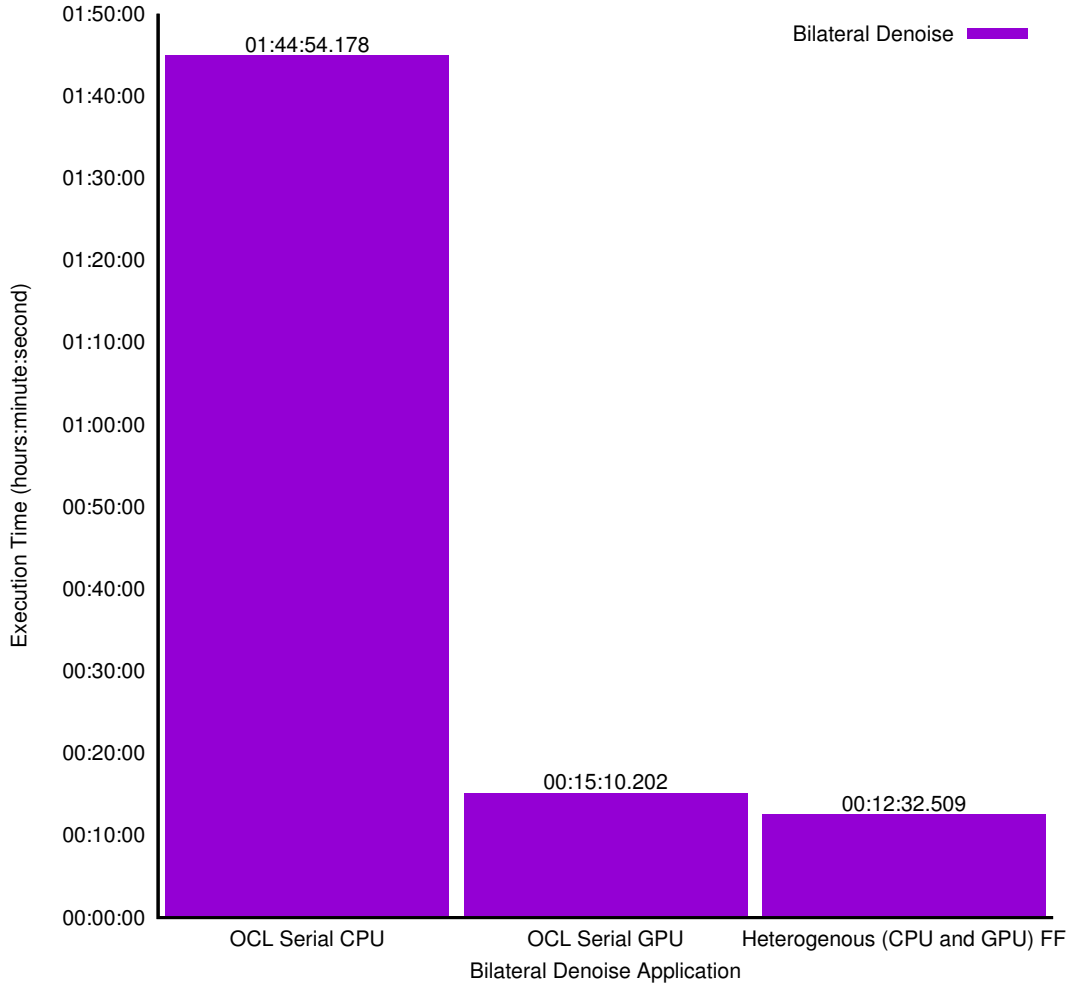


Figure 5.8: Execution of Bilateral denoise using OpenCL back-end

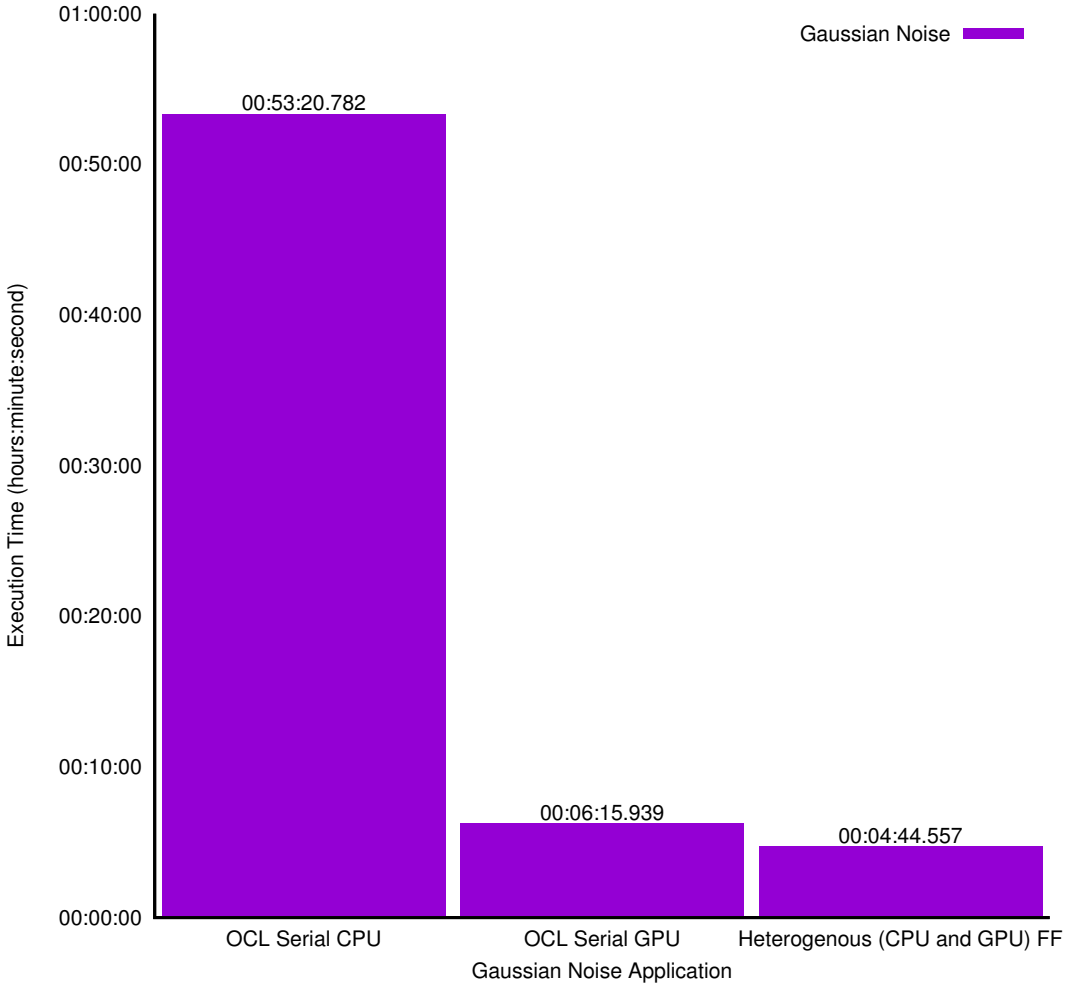


Figure 5.9: Execution of Gaussian noise using OpenCL back-end

OpenCL Component Tuning: To clarify the parallel scaling of an OpenCL component on a different device, we have executed the bilateral application with different group size configurations for a CPU-allocated OpenCL worker. The application receives streams of 5120 different images sizes an inputs ranging from 64*64 to 2048*2048, presented in table 5.4.

The first three columns of the table represent the bilateral application with only one CPU-allocated worker for the second farm stage. As demonstrated in Figure 5.10, when the only CPU-allocated worker is applied, exhausting all 12 CPU cores is faster than using 4 or 8 cores.

Columns 4, 6 and 7 represent the execution time of the bilateral application when the GPU-allocated worker is added to the farm stage. Comparing the three columns, the execution time of 1 GPU-allocated OpenCL worker and 1 CPU-allocated OpenCL worker using 8 cores is faster than that using all 12 CPU cores. Moreover, the execution time when using 12 cores is even slower than using a farm with only one GPU-allocated OpenCL worker.

Monitoring the system during execution reveals that when a GPU device is used for an application with streams of input data, at least one host thread is required to constantly feed the GPU worker. Exhausting all cores leads the GPU-allocated host thread to compete to acquire the CPU-resources. This will interrupt the GPU-allocated components being fed and the GPU utilisation drops as a result of waiting for an input because the feeding thread is competing for resources. Therefore, when both the CPU and GPU-allocated OpenCL workers are applied, exhausting all cores can have a negative effect on application performance.

5.2.3 Workload Distribution

In this section we evaluate the efficiency of HFastFlow adaptive load-balancer presented in Chapter 4, section 4.1.2 and the adaptive workload distribution technique presented in Chapter 4, section 4.4.2 to a tune the proposed load-balancer.

Table 5.6 shows the workload fraction for the bilateral denoise application for the streams of 6144 images of 6 different size categories presented in Table 5.4. The application has been executed on a node of the Xookik cluster. The farm stage in the application has two OpenCL workers. The second and third columns represent the average workload fraction of the CPU-allocated and GPU-allocated OpenCL workers for different categories of the images respectively.

Table 5.7 demonstrates the detailed execution time of the bilateral denoise application for the 1024*1024 category.

When the HFastFlow original load-balancer is applied, limiting the queue size can adjust the number of tasks executed by each worker that corresponds to the worker service time. The workload fraction varies depending on the size of the input images.

Using unbounded queues for workers with an original load-balancer demonstrates a significant drop in application performance. Comparing the last row of the table with the first one

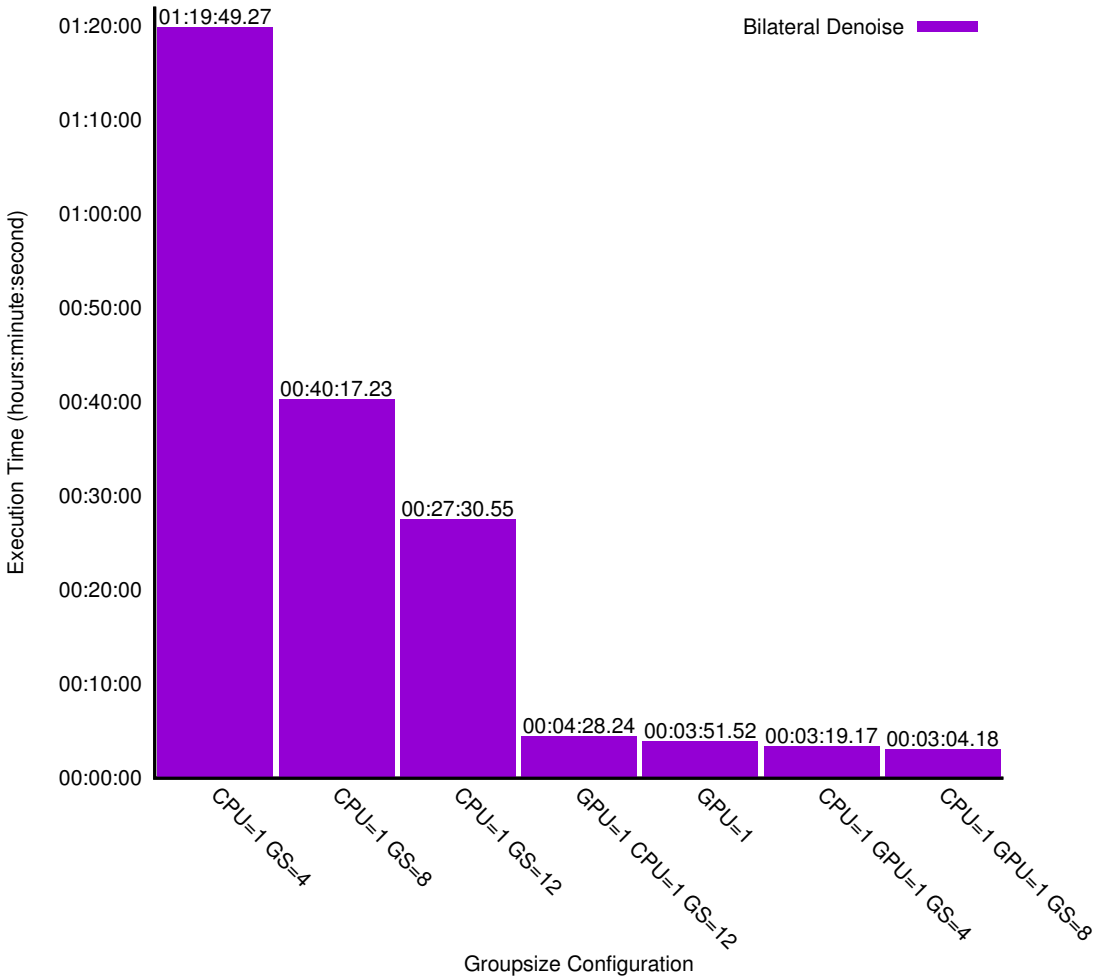


Figure 5.10: Execution of Bilateral denoise using different group-size (GS) for CPU-allocated OpenCL workers on a node of the Xookik cluster. The number of CPU or GPU workers used for each execution is demonstrated in the caption of each column.

Table 5.6: Workload fraction in proportion with the computing power for each OpenCL component of the Bilateral Denoise

Task-Input Size	CPU-allocated worker	GPU-allocated worker
64*64	0.016497	0.983503
128*128	0.103976	0.896024
256*256	0.082229	0.917771
512*512	0.075479	0.924521
1024*1024	0.074882	0.925118
2048*2048	0.073479	0.926521

determines that when a GPU-allocated worker is only applied, the execution time of the application is 9 times faster than the one using both workers with an unbound queue and original load-balancer.

Moreover, for the original load-balancer the execution time of the application varies depending on the queue size. As stated in the third row of the table, the execution time of the application using the original load-balancer is slower than the execution time of the application that only has one GPU-allocated OpenCL worker (the last row).

Considering the execution time of the adaptive load-balancer (the fourth row) demonstrates that the execution time of the application for the workers with unbounded queues is intact.

Comparing the original load-balancer with the proposed adaptive one states that, for the original load-balancer, an application performance depends on the queue size, while, our adaptive load-balancer application performance depends on the input size.

Adaptive Workload Distribution: Applying divisible workload theory dynamically adjusts the fraction of the workload when the weight-based policy is applied.

Figure 5.11 demonstrates an *ad-hoc* switching mode policy where the sudden changes in the input stream size immediately affect the workload distribution variation. As stated in Figure 5.11, there are fluctuations in the figure, which we refer to as *transient states*. Considering any changes in the size of data as a milestone in the application execution time, these transient states are the result of an asynchronous meeting of the milestones by each worker. When the faster worker reaches one milestone while the slower one is still running the input stream of the previous milestone, the service time based calculated workload fraction will be inaccurate until the slower one reaches the milestone. The ad-hoc policy is useful when a worker reallocation is required in the scheduling policy. In this case, a sudden change is required to readjust the workload distribution according to the new service time of the worker.

Table 5.7: The bilateral application execution and different runtime for each component

Load-balancer policy	Total execution time	CPU-allocated OpenCL worker execution time	GPU-allocated OpenCL worker execution time	Total assigned tasks to CPU-allocated OpenCL worker	Total assigned tasks to CPU-allocated OpenCL worker
FF_Loadbalancer- queue limitation-unbound	906.855	906.370	72.040	512	512
FF_Loadbalancer- queue limitation-1	135.770	135.142	133.450	75	949
FF_Loadbalancer- queue limitation-10	150.168	149.856	132.248	81	943
Adaptive_Loadbalancer- queue limitation-unbound	135.183	134.556	133.520	74	950
Farm with only one GPU-allocated OpenCL worker	144.433	0	143.855	0	1024

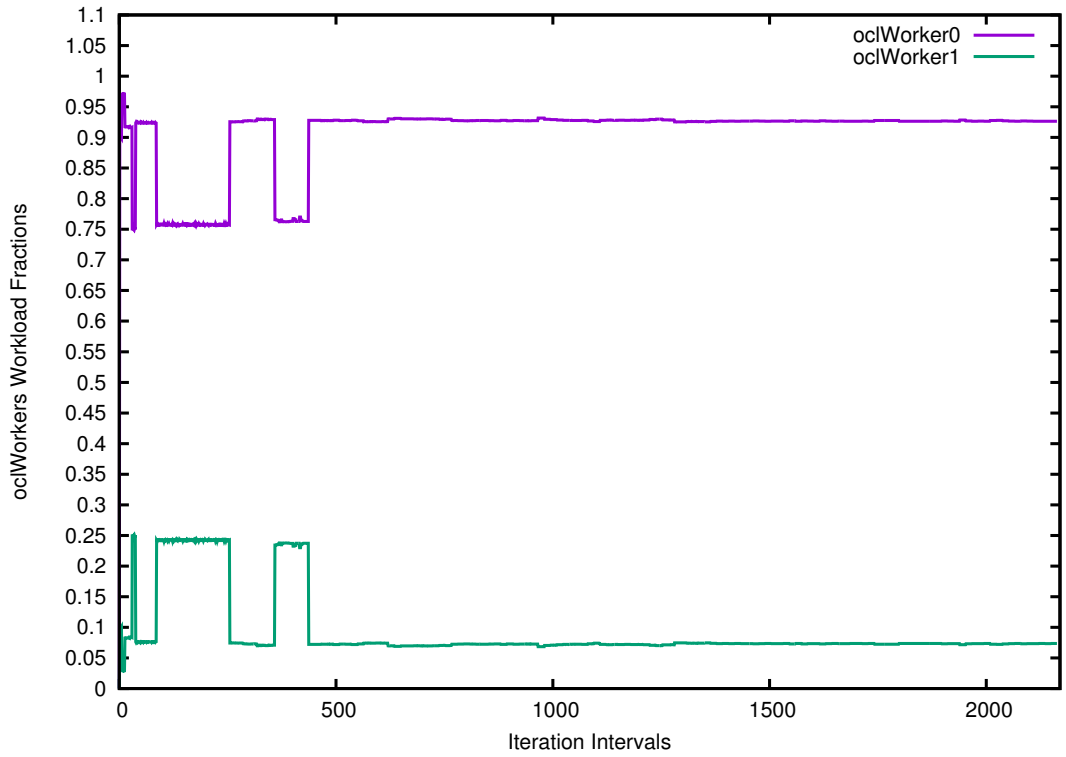


Figure 5.11: The variation of workload distribution for CPU/GPU workers for different input stream sizes on a node of the Xookik cluster using the ad-hoc policy for bilateral denoise

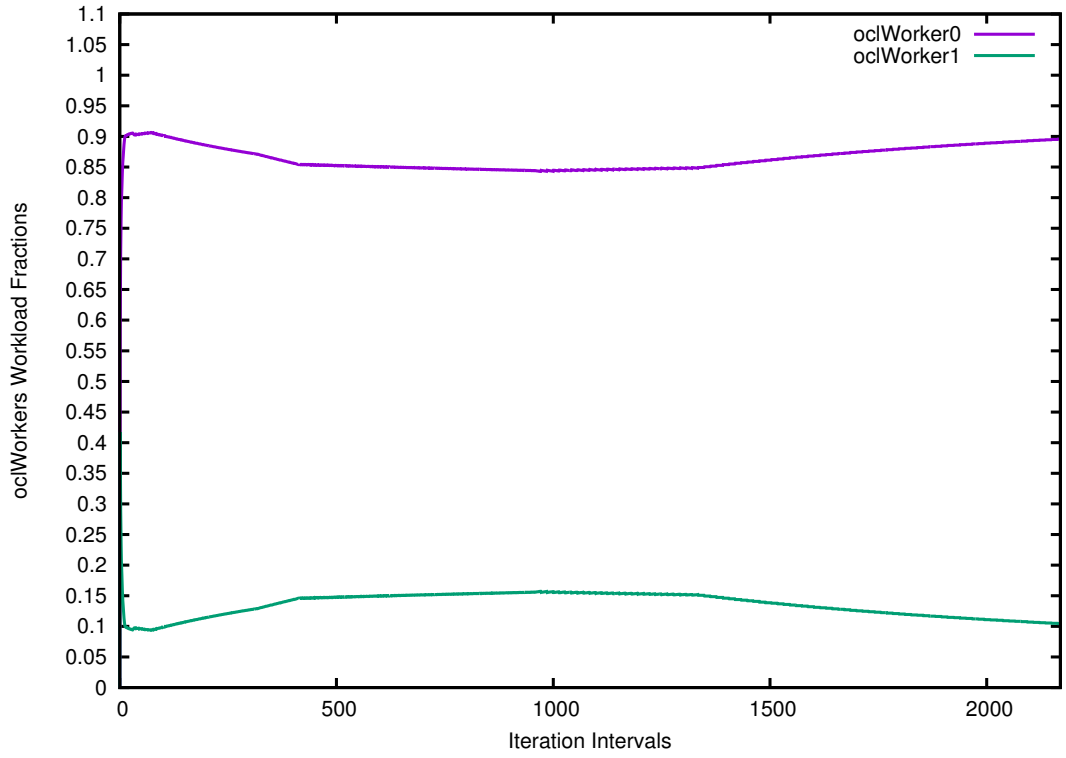


Figure 5.12: The variation of workload distribution for CPU/GPU workers for different input stream sizes on a node of the Xookik cluster using the average policy for bilateral denoise

Figure 5.12 demonstrates the workload distribution using the *average* policy. In the average policy, the average service time of a worker since the beginning of an application execution is considered to calculate the workload fraction. In comparison with Figure 5.11, the average policy has a smooth movement when the input stream size changes. This would be useful when resource reallocation is not required. In the absence of resource reallocation, the computational power of the nodes is predictable and a moderate refinement is required over the workload distribution according to the new image size.

5.2.4 Phase Changing Prediction

In this section we evaluate the efficiency of the phase changing prediction techniques of the proposed OpneCL scheduler presented in Chapter 4, section 4.4.3.

We have applied the resource stealing and resource sharing techniques in the bilateral denoise and recursive Gaussian applications respectively.

Resource Stealing: As stated in the section 5.2.3, when the original load-balancer in HFast-Flow is applied, an application performance depends on the queue size, while, when our weight based load-balancer is applied, an application performance depends on the input size.

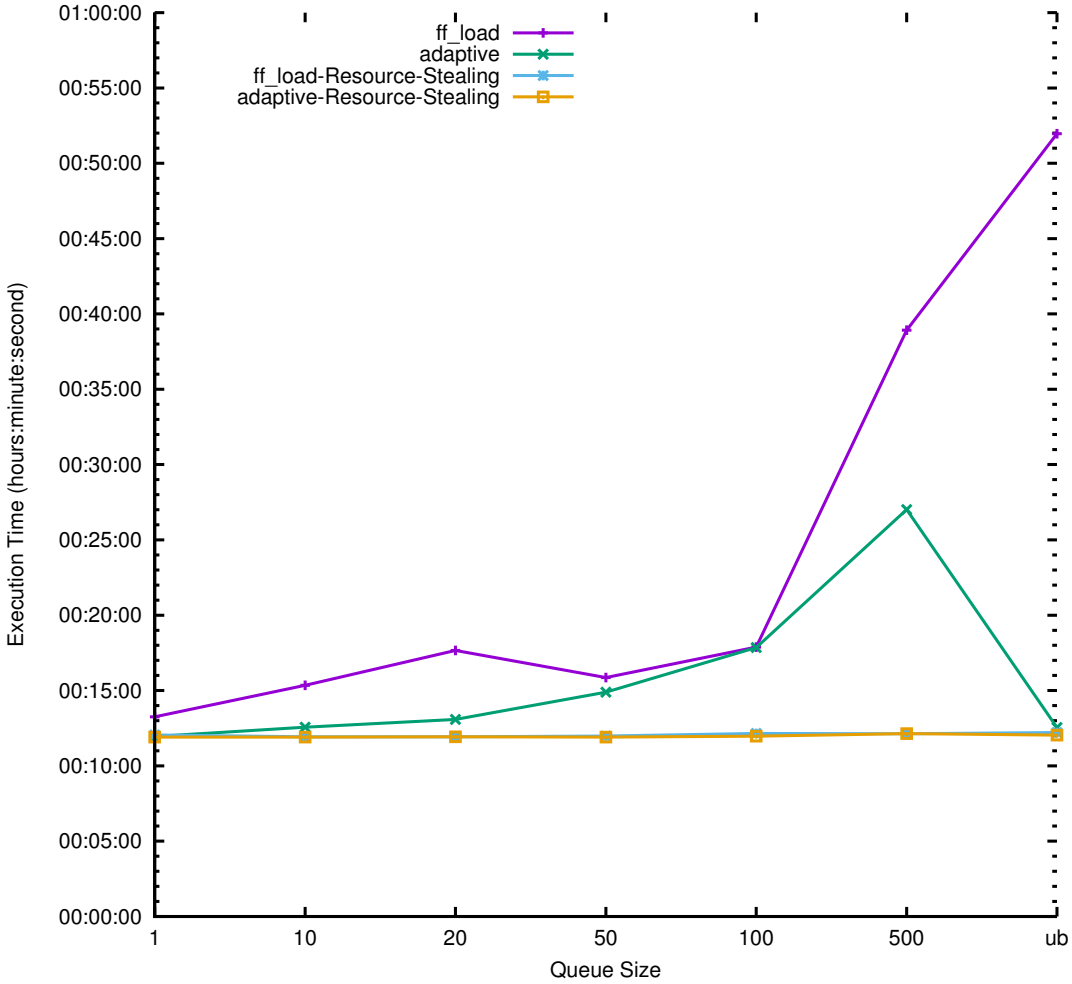


Figure 5.13: The execution time of the bilateral denoise application for different load-balancers, different queue sizes and different input image stream sizes on a node of the Xookik cluster using the average policy.

Figure 5.13 demonstrates four different executions of the bilateral denoise application for both the round robin load-balancer and the adaptive load-balancer with and without resource stealing.

In order to determine the dependency of the adaptive load-balancer to input size variation we have applied the streams of 6144 images of 6 different size categories presented in Table 5.4. Also, the application has been executed for different queue sizes in order to demonstrate the dependency of the round robin load-balancer on different queue sizes.

Comparing the gradient of the contour line of the original load-balancer with that of the adaptive one demonstrates that the dependency of the original load-balancer on the worker queue size precipitates an application performance drop that is greater than the dependency of the adaptive load-balancers to the input size.

When our resource stealing technique is applied, comparing the results for both load-balancer policies states that the execution time of the application for different queue sizes and image sizes is almost the same. Therefore, applying resource stealing makes the workload distribution policies independent from queue size and task size.

Resource Sharing: Figure 5.14 represents the execution time of the recursive Gaussian application for different component allocation policies. The application receives the streams of 6144 images of 6 different size categories presented in Table 5.4 and the average policy is applied for workload distribution. The recursive Gaussian application is a seven stage pipeline composed of four OpenCL stages: two RG operators and two transpose stages.

To schedule these four OpenCL nodes 4 different strategies are considered.

Considering the order of the devices for the node of the xookik cluster that is used here, the first OpenCL resource is the CPU and the second is the GPU. Therefore, by applying the round robin policy, the first RG stage is allocated to the CPU; the second stage is allocated to the GPU; the third is allocated to the CPU; and the fourth is allocated to the GPU.

The greedy technique tries to allocate as much as it can to the GPU device. When there is no slot available on the GPU, it will allocate the device to the CPU. Therefore, applying the greedy algorithm, the first two OpenCL stages are allocated to the GPU and the second two are allocated to the CPU device.

The static structural configuration tries to find an optimum solution by recalculating the component augmentations and allocation. Using the reduction rules provided in equation 4.12, the abstract computation graph can be generated as follows:

$\wedge$«*readstage*»$\wedge$ «||gausianFilter||» $\wedge$«||transform||»$\wedge$ «||gausianFilter||» $\wedge$ «||transform||» $\wedge$«*readstage*» $\wedge$ «*readstage*»

Figure 5.15-a represents the visualisation of the abstract computation graph for recursive Gaussian and Figure 5.15-b demonstrates the MCTS decision tree generated for recursive

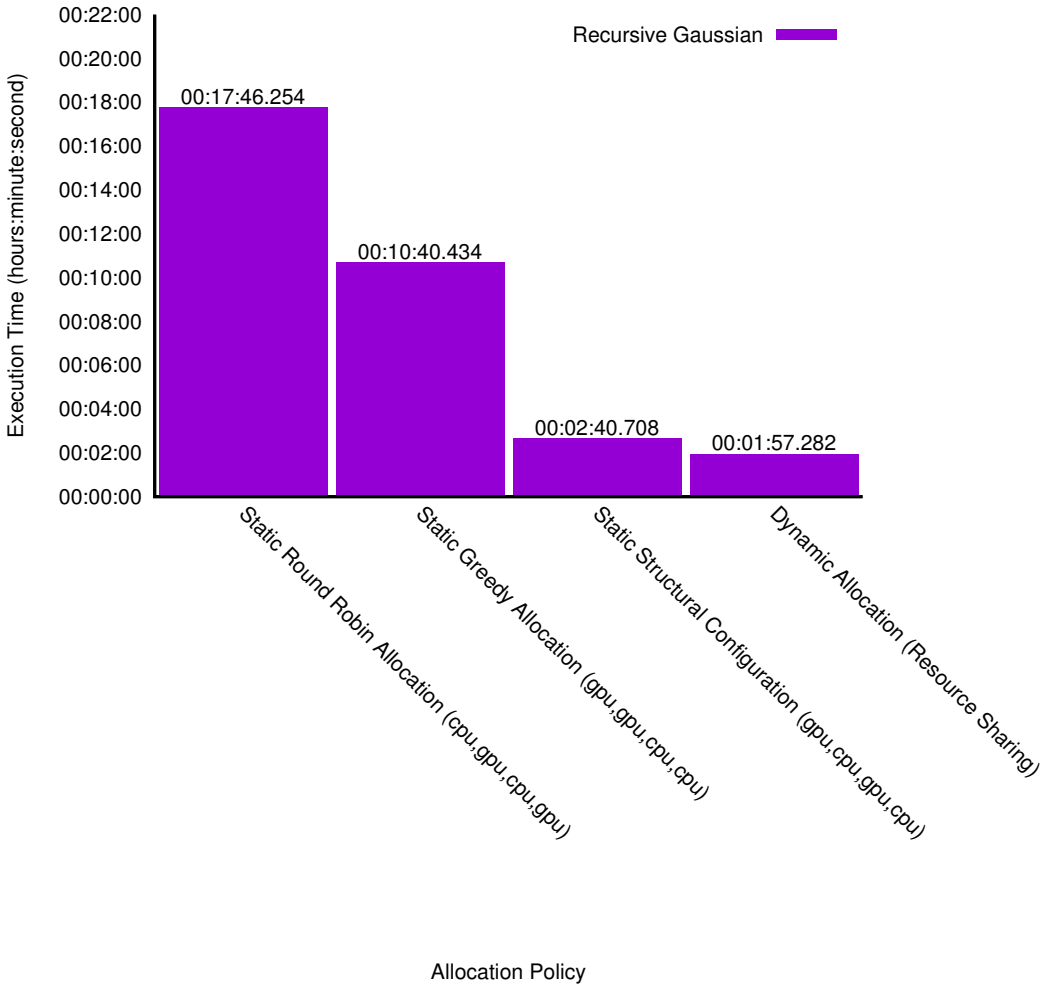
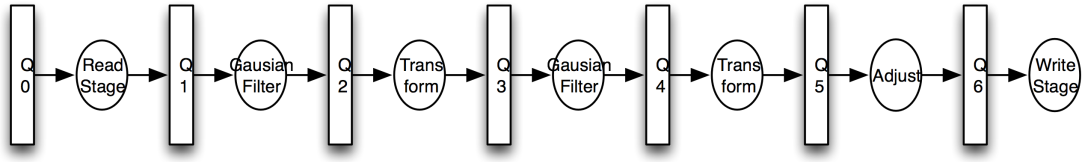
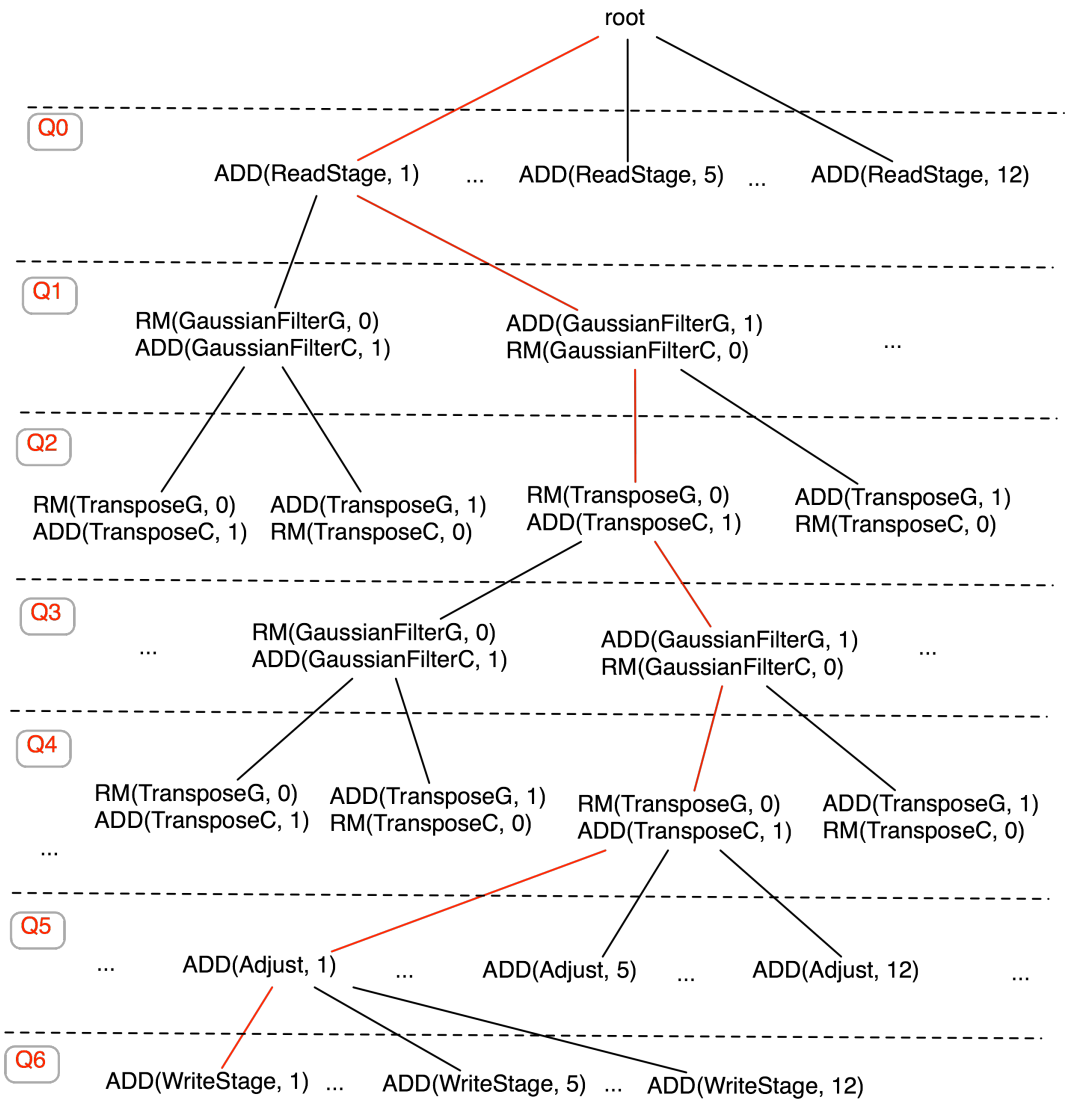


Figure 5.14: The execution time of the recursive Gaussian application for different allocation policies on a node of the Xookik cluster



a)



b)

Figure 5.15: a) Abstract computation graph for recursive Gaussian. b) MCTS decision tree generated for graph a

Gaussian. For all OpenCL components in the MCTS decision tree, two separate entries for CPU and GPU have been considered, as they can be executed on both CPU and GPU. By applying the capacity constraint on OpenCL devices, none of the OpenCL stages can have more than one OpenCL component. Therefore, for each OpenCL component, finding the best resource to allocate is the challenge. In this case $12 \times 2 \times 2 \times 2 \times 2 \times 12 \times 12 = 27648$ possible solutions are available.

Using the MCTS algorithm, the structural configuration and mapping of the component to the available resources for the node used in the Xookik cluster is:

$$
\begin{aligned}
&\{ \\
&\quad readBenchmark = 1; \\
&\quad RG1_{CPU} = 0; RG1_{GPU} = 1; \\
&\quad trans1_{CPU} = 1; trans1_{GPU} = 0; \\
&\quad RG2_{CPU} = 0; RG2_{GPU} = 1; \\
&\quad trans2_{CPU} = 1; trans2_{GPU} = 0; \\
&\quad adjust = 1; \\
&\quad writeBenchmark = 1 \\
&\}
\end{aligned}
\tag{5.1}
$$

The resource sharing technique is a dynamic approach which reallocates the components according to a dynamic change in the bottleneck node.

As stated in figure 5.14 by applying the resource sharing technique over recursive Gaussian, we have achieved almost 1 order of magnitude speed-up over the round robin technique; 5 times speed-up over the greedy algorithm; and a 28% performance improvement over the static skeleton configuration.

5.2.5 Multi-tenant Application

In this section we evaluate the efficiency of the proposed OpneCL scheduler presented in Chapter 4, section 4.4.3 to support both concurrent and parallel execution of multi-tenant applications.

Parallel Execution of Multi-tenant Application: Tables 5.9 and 5.8 represent the parallel execution of a sobel and URNG applications over the Titanic machine and a node of the Xookik cluster respectively. The capacity constraint for each OpenCL device is set to one.

Each application has one OpenCL stage. The allocation of OpenCL devices to each application varies depending on the priority of the application.

The first two rows in each table represent the execution of each application with static OpenCL resource allocation regarding the applications' priorities. No dynamic reallocation is

Application Composition	SOBEL-EXEC-TIME	URNG-EXECTIME	TOTAL-EXECTIME
"Sobel-CPU, URNG-GPU"	00:29:06.124	00:02:44.148	00:29:06.124
"Sobel-GPU, URNG-CPU"	00:02:09.536	01:17:46.220	01:17:46.220
"Soble-GPU, URNG-CPU/GPU"	00:02:45.655	00:04:49.260	00:04:49.260
"Sobel-CPU/GPU, URNG-GPU"	00:04:41.727	00:02:44.023	00:04:41.727

Table 5.8: The runtime for concurrent execution of the Sobel Filter application and URNG on a node of the Xookik cluster

Application Composition	SOBEL-EXEC-TIME	URNG-EXECTIME	TOTAL-EXECTIME
"Sobel-CPU, URNG-GPU"	01:34:08.420	00:03:30.792	01:34:08.420
"Sobel-GPU, URNG-CPU"	00:02:48.640	03:33:13.060	03:33:13.060
"Soble-GPU, URNG-CPU/GPU"	00:02:48.668	00:06:14.493	00:06:14.493
"Sobel-CPU/GPU, URNG-GPU"	00:06:13.457	00:03:30.942	00:06:13.457

Table 5.9: The runtime for concurrent execution of the Sobel Filter application and URNG on the Titanic machine

involved for the first two rows.

On both the node of the Xookik cluster and the Titanic machines, two different priority configurations have been considered. The first row in each table represents that the URNG priority is higher than the sobel filter priority. The second row in each table represents that the sobel filter has more priority than the URNG application.

The third and fourth rows in each table represent priority based resource allocation by using the OpenCL scheduler with regards to dynamic reallocation. With the ability to adaptively respond to the environmental change, the OpenCL scheduler will dynamically reallocate the lower priority component when the higher priority application terminates. The result of the adaptive scheduling represents a significant improvement in the total execution time of the applications.

Considering the difference in computational power of the CPU and GPU for executing the OpenCL kernel in our cases, executing multi-tenant applications in parallel mode is different from normal parallel execution where all resources are computationally equivalent.

Considering the two columns in both Tables 5.9 and 5.8 represents that using both CPU and GPU in parallel when they are not computationally equivalent is slower than executing them serially on the GPU.

However, applying the phase detection technique (the last two columns in both Tables 5.9 and 5.8) will solve the problem. In this case, the applications use the GPU concurrently while

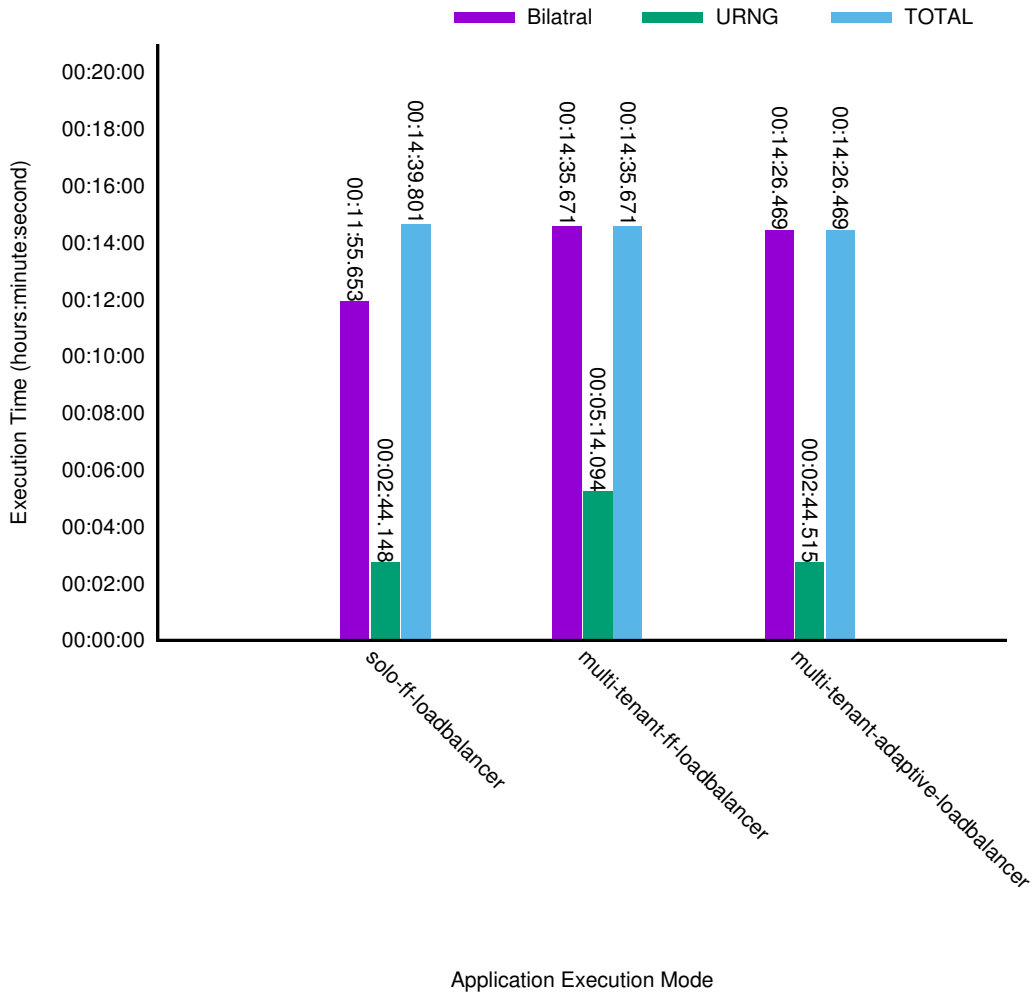


Figure 5.16: The concurrent execution of the URNG and Bilateral denoise applications where the priority of the URNG application is higher than the priority of the Bilateral denoise application

they run in parallel. Therefore, the CPU is used to assist the GPU device, not to replace it.

Concurrent Execution of Multi-tenant Application: Figure 5.16 demonstrates both the serial and concurrent executions of URNG and Bilateral denoise applications. For the concurrent execution, the initial priority of the URNG is higher than initial priority of the bilateral denoise.

Also, Figure 5.17 represents both the serial and concurrent execution of the convolution and sobel filter applications. For the concurrent execution, the initial priority of the sobel filter is higher than the initial priority of the convolution application.

The capacity constraint for each OpenCL device is set to one and the priority policy is variable. In each figure, the first category represents the serial execution of the applications; the second category represents the concurrent execution of the applications using the appli-

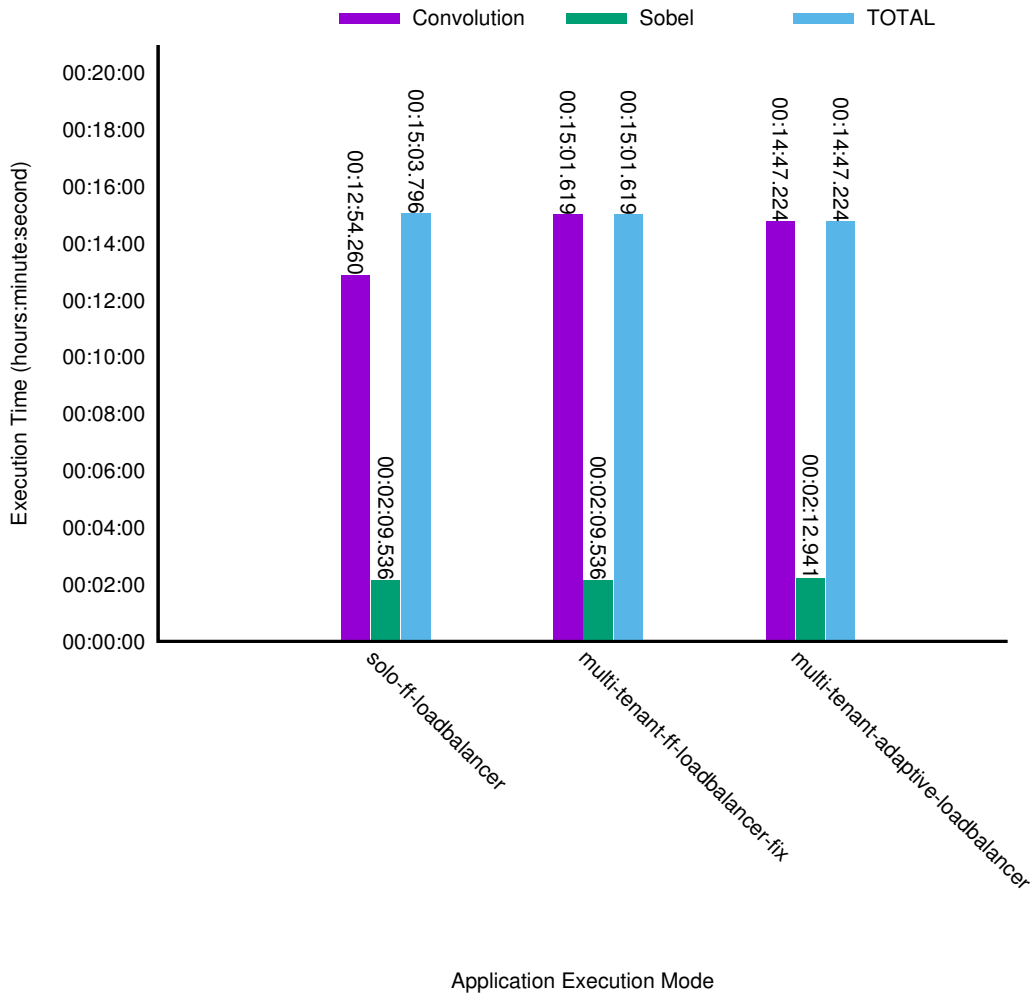


Figure 5.17: The concurrent execution of the sobel filter and convolution applications where the priority of sobel filter application is higher than the priority of the convolution application

cation suspending technique and the third category represents the concurrent execution of the application using the component masking technique.

As the masking technique does not affect the application priority, there is no change in the execution order of the application when the component masking technique is applied. In this case, the result for both the variable and fix priorities is the same. The variable priority policy for the suspending technique ensures a fair distribution of resources among the applications. This claim is proof as the execution time of the URNG and sobel filter application are almost double and the sums of the execution times of the two mentioned applications have been added to the bilateral and convolution applications respectively.

Moreover, comparing the concurrent execution time with the serial version demonstrates that the overhead of switching among applications to share the resource is negligible. Also, the masking technique states up to 2% improvement on the total execution time.

Execution Demonstration Figure 5.18 demonstrates the procedure of executing the convolution application in concurrent with the sobel filter where the adaptive workload distribution with the ad-hoc policy, the masking technique and the resource stealing technique for the OpenCL farm in a separable convolution are applied.

The execution procedure is divided into four stages.

The first stage is the solo execution of the convolution application where one worker has been assigned to the CPU and one worker has been assigned to the GPU (interval 0 to 10). As stated in Figure 5.18, the GPU-assigned worker is faster than the CPU-assigned one and receives around 85% of the input workload, while the slower one receives the remaining 15% of the workload.

The second stage starts from interval 10 to 300, which is the time when the sobel Filter application is executed. As the sobel filter priority is higher than that of the convolution, it wins the competition for the GPU resource. Using the masking technique, the GPU-assigned worker of the bilateral denoise is masked and all the input workload is redirected to the CPU-assigned one.

The third stage starts from time interval 300 to 1020, when the sobel filter execution terminates. In this stage the GPU-assigned worker is unmasked and the workload distribution is readjusted for both workers. Using an ad-hoc policy for the workload distribution, the transition states in the figure represent the asynchronous accessing of different input data sizes.

The fourth stage starts from time interval 1020, where the GPU-assigned worker has exhausted all its assigned tasks. The resource stealing method is invoked by using the phase changing detection in the OpenCL scheduler and reallocates the CPU-assigned worker to the GPU.

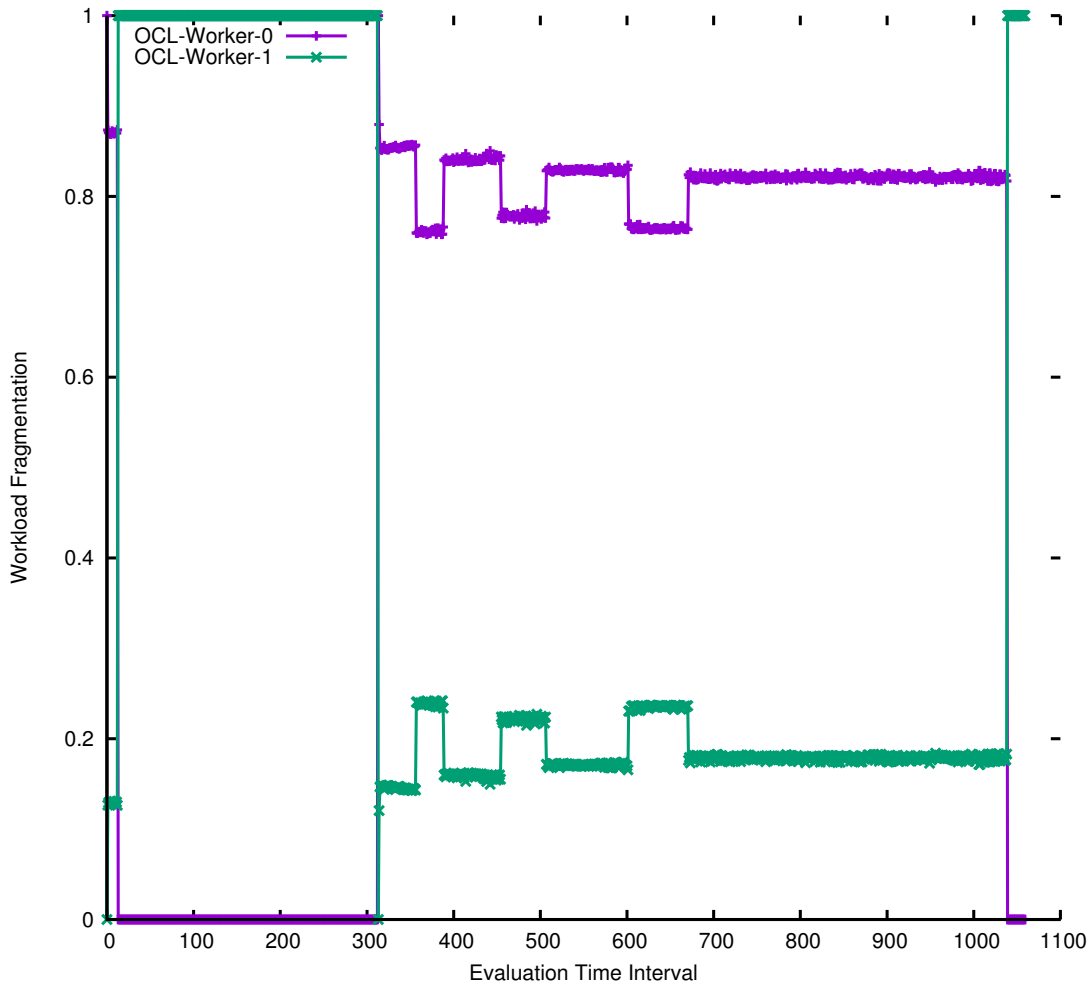


Figure 5.18: The visual demonstration of the concurrent execution intervals for the sobel filter and convolution applications where the priority of sobel filter application is higher than the priority of the convolution application

5.3 Summary

In this chapter we have designed and developed structured applications using PEI for tuning coordination over heterogeneous multi-core architectures. In section 5.1, we have demonstrated the feasibility of using the HFastFlow SKIP adaptor for generating structured applications using different combinations of HFastFlow patterns.

In section 5.2, following evaluations have been performed to assess the adaptability and performance improvements of PEI based applications using our OpenCL back-end. The evaluation have been carried out on two different heterogeneous multi-core architectures, namely, RGU Xookik cluster and the university of Pisa Titanic machine specified on Table 5.3.

Performance Overhead: In section 5.2.1, we have demonstrated that the minimum and maximum overheads of the PEI framework for the proposed image processing applications are 1% and 2.5%, respectively. These overheads are related to the cost of extracting the sensor information dynamically with regards to the applied monitoring policy. Also, the minimum overhead can be used to determine a lower bound for improvements that should be achieved by the coordination optimisation methods.

OpenCL back-end: In section 5.2.2, by using the OpenCL back-end with PEI coordination engines, we have achieved 38% improvement on GPU utilisation for the Sobel filter application. We have also achieved up to 1.63 times speed-up over the serial version of Sobel filter executing on a GPU-device. For both bilateral denoise application and Gaussian noise application, we have achieved up to 9 times speed-up over the serial version of the application on CPU. We have also achieved up to 1.21 times speed-up for both applications over theor serial versions executing on a GPU-device.

Workload distribution: In section 5.2.3, by using the adaptive workload distribution on bilateral denoise application, not only we have removed the dependency between the component queue-size and application performance that causes up to 9 times drop in application speed-up for unlimited queue size, but we have also achieved up to 7% performance improvement. However, the adaptive load-balancer used for bilateral denoise application demonstrates dependency to data input size when it is changed dynamically.

OpenCL Scheduler: In section 5.2.4, by using the resource stealing techniques provided by the OpenCL scheduler, we have removed the dependency between the component queue size and input data size for both HFastFlow original load-balancer and our adaptive load-balancer respectively. Moreover, by applying the resource sharing technique to allocate component on OpenCL-enable devices for recursive Gaussian application, we have achieved up to one order of magnitude speed-up over HFastFlow round-robin technique and up to 5 times speed up over static structural configuration.

Multi-tenant application: In section 5.2.5, we have evaluated the efficiency of using our OpenCL scheduler for executing multiple applications both in parallel and concurrent mode. We have considered different combinations of application executions, namely:

- Parallel execution of Sobel filter and URNG, where the OpenCL node for one application is allocated to CPU and the OpenCL node for the other application is allocated to GPU.
- Concurrent execution of Sobel filter and URNG where both application share the CPU and GPU resource based on their priority.
- Concurrent execution of Sobel filter and simple-convolution where both application share the CPU and GPU resource based on their priority.
- Concurrent execution of URNG and bilateral denoise where both application share the CPU and GPU resource based on their priority.

Parallel executions of Sobel filter and URNG applications have demonstrated that the total execution time of applications are slower than executing them serially on GPU. However, the concurrent executions of applications have demonstrated that the overhead of switching resources between components is negligible. Moreover, we have achieved up to 2% improvement on total execution time of application.

Although executing multi-tenant applications in a concurrent mode may not significantly improve the performance, it provides extra flexibility for the applications' executions. In this case, by prioritising applications, small-scale applications can temporarily borrow resources from the large-scale ones without terminating the execution of the large-scale applications. Applying the phase detection technique, it is possible to use CPU to assist the GPU device, however, it is not a replacement for a GPU.

Chapter 6

Evaluation of Generic Applications

In this chapter we evaluate PEI for existing FastFlow applications which do not use *HWrapper* to evaluate the generality of the proposed methodology. These applications are mainly the GPU/CPU based numerical algorithms which are suitable for parallelism. These applications have already been implemented without using the SKIP adaptor. The evaluation of the existing applications is provided to demonstrate that with no modification of existing applications, it is possible to benefit from the provided coordination optimisation by using SKIP-based instrumented building blocks instead of the default building blocks.

Two categories of applications have been considered here. The first category executed on a homogeneous architecture containing a set of benchmark applications provided by the FastFlow developer team. As benchmark applications for FastFlow, these applications are selected to demonstrate the overhead of the SKIP methodology in a building block based framework like FastFlow. Also, we demonstrate the efficiency of the SKIP compliant building block based approach for coordinating homogeneous applications. Moreover, we have implemented the *N*-body simulation in three different frameworks to compare the overhead of the building block based approach in application scalability in comparison with the highly specialised skeleton based approach.

The second category of applications has been deployed in the hybrid (CPU/GPU) system using the *ParWrapper* provided in FastFlow. These applications are composed of a set of GPU and CPU components. The GPU component for these applications is implemented in the CUDA environment. As these applications are using *ParWrapper* instead of *HWrapper*, not all coordination engines can be applied in these applications. However, using an instrumented version of building blocks in the FastFlow framework, we indicate that our *HWrapper*-independent coordination methods can efficiently optimise these applications. The second category of applications has been chosen carefully to demonstrate *i)* the stability and adaptability of the SKIP-compliant building block approach that is embedded as part of a large scale distributed intense memory usage application using the applicable coordination engines (*EI* Routine); and, *ii)* the

generality of certain coordination engines applicable in heterogeneous applications not using the *HWrapper* extension (SMTWTP and MD). We have applied the structural tree notations presented in Table 5.2 to indicate the architectural view of each application in FastFlow.

6.1 Homogeneous Application

This section contains six different FastFlow existing applications that do not contain any *HWrapper* or *ParWrapper* components. Therefore, GPU is not used when executing these applications on heterogeneous multi-core architectures. By applying these applications we demonstrate the efficiency of PEI for executing applications on Homogeneous (CPU only) multi-core architectures.

Table 6.1 represents the characteristics of each application. The first column of the table represents the application name and the second column represents the features that the application covers.

Table 6.1: Summary of Homogeneous Applications Characteristics.

Application Name	Application Characteristics
N-body-Simulation	The serial version of *N*-body simulation is implemented as a benchmark application by SICSA multi-core challenge [103]. We have implemented *N*-body-Simulation on FastFLow, SKePU, and Thrust to evaluate the overhead of RISC-Pb^{2}l building block based framework with other existing algorithmic skeleton based frameworks. This is an example of embedding the FastFlow framework in an application in order to parallelise it. The *N*-body-simulation application in FastFlow is the concatenation of 3 Farm patterns with barrier, where each Farm is parallelising an independent block of codes in *N*-body-Simulation.
Mandelbrot	These applications are implemented by FastFlow development team as standard benchmark applications. They represent the usage of FastFlow Farm pattern for parallelising different numerical applications on homogeneous (CPU only) multi-core architectures. In this thesis, we have used these applications to evaluate the overhead of PEI framework for extracting/injecting sensor/actuator information. Moreover, the efficient idling coordination technique presented in Chapter 4, section 4.1.4 is applied to these applications to evaluate the efficiency of PEI framework for auto-tuning the dynamic coordination of applications that do not use any *HWrapper* or *ParWrapper* components.
Quick Sort	
Fibonacci	
Stencil	
N-queens	

In the following we introduce them in brief.

6.1.1 N-body Simulation

The N-body problem is a classical problem of predicting the individual motions of a group of celestial objects that interact with each other gravitationally.

The application has the 4 following steps:

- Calculates the initial momentum of the bodies in the system;
- Calculates the energy which is a combination of the kinetic energy between each pair of bodies and the potential energy when the bodies are far apart.
- Calculates the motions of bodies for S step iterations.
- Recalculates the total energy of the system after the motion to demonstrate that the total energy is constant.

At the end of each step a synchronisation mechanism is required to make sure each step is applied for all bodies before applying the next step. However, the operation of each step in different bodies can be executed independently. In this case the application is composed of 4 separate parallel parts where at the end of each step a barrier is applied to synchronise the operation.

Using the building block grammar represented in Listing 3.1, each step for the N-body simulation is generated as follows:

First step: Initial momentum $\square_{Momentum}$ presents a composition that calculates the initial momentum of the bodies in the system.

Using RISC-pb^{2}l building block notations, we have designed the momentum step as follows:

$(f_{momentum_{1 \lhd n}}) \centerdot [\text{«momentom»}]_n \centerdot (g_{momentum_{n \rhd 1}})$

Second step: Total energy before S step motions $\square_{Energy}$ shows a composition that calculates the total energy of the system before and after the movement.

Using RISC-pb^{2}l building block notations, we have designed the energy step as follows:

$(f_{energy_{1 \lhd n}}) \centerdot [\text{«energy»}]_n \centerdot (g_{energy_{n \rhd 1}})$

Third step: S step motions $\square_{Advance}$ indicates a composition that calculates the motion of the bodies towards the initial position after s step iterations. It has been encapsulated inside a `for loop` to calculate the motion of all bodies for s steps.

Using RISC-pb^{2}l building block notations, we have designed the motion step as follows:

$(f_{advance_{1 \lhd n}}) \centerdot [\text{«advance»}]_n \centerdot (g_{advance_{n \rhd 1}})$

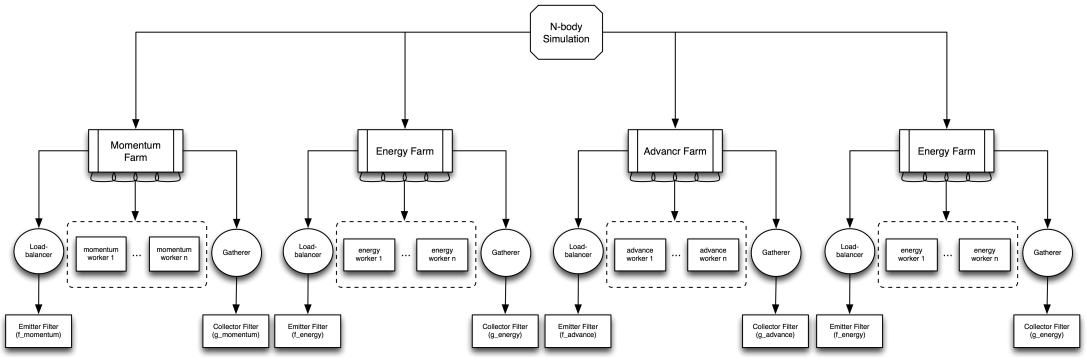


Figure 6.1: The structural composition of components for the N-body simulation in FastFlow visualised by structural tree notation

Fourth step: Total energy after S step motions The same composition generated for the second step is applied on the bodies to calculate the total energy after s step motions.

Appendix D.1 represents the generation of each pattern through the RISC-pb^{2}l building block presented in Listing 3.1. Replacing the serial implementation of each step with its correspondent generated pattern, we have embedded PEI inside the N-body simulation to parallelise the application. Figure 6.1 represents the architectural view of the N-body simulation that encapsulates the FastFlow parallel patterns.

In FastFlow for each composition a separate farm pattern has been implemented and embedded inside the applications.

For the farm patterns in *Momentum* and *Energy* an instance of `ff_loadbalancer` with an *emitter* filter function has been applied which divides the bodies into chunks of K size where $K = N/nworkers$; N is the number of bodies and $nworkers$ represents the number of farm workers. This will represent a scatter with a custom division of data. Also, an instance of `ff_gatherer` with a *collector* has been used to reduce each property value to a scaler number.

The farm pattern for *Advance* is also equipped with the `ff_farm::wrap_around()` method to implement the feedback. An instance of `ff_loadbalancer` with an *emitter* filter function is used as a load-balancer filter which splits bodies into square blocks and offloads them to workers' queues asynchronously by using the round-robin technique. Each worker concurrently calculates the force of the received square tiles so that the interactions in each tile are evaluated in sequential order using the upper triangular technique, thus generating the partial updated vector. Also, an instance of `ff_gatherer` with a *collector* is used to collect the partial updated vector from workers and this updates the bodies.

For each farm pattern generated in FastFlow, the workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_freeze()` method. The method `ff_farm::wait_freezing()` invokes the barrier required for synchronisation at the end of each farm before going on to the

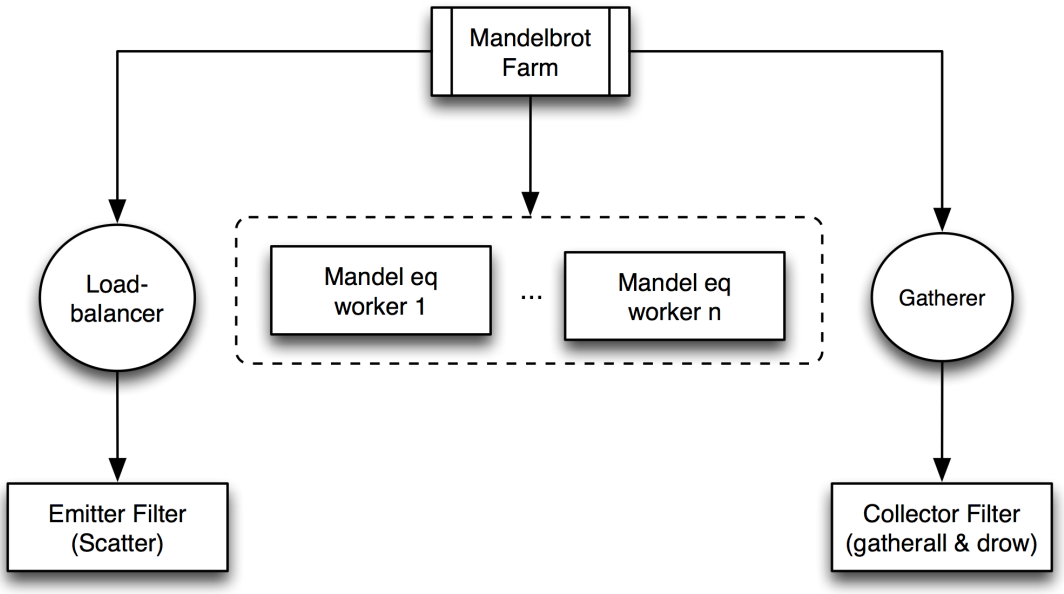


Figure 6.2: The structural composition of components for the Mandelbrot application in FastFlow visualised by structural tree notation

next step.

6.1.2 Mandelbrot

The Mandelbrot set is the set of values of c in the complex plane for which the orbit of 0 under the iteration of the complex quadratic polynomial

$$z_{n+1} = z_n + c \tag{6.1}$$

remains bounded [104]. When starting with $z_0 = 0$ and applying the iteration repeatedly, a complex number c is part of the Mandelbrot set if the absolute value of z_n remains bounded however large n gets. The *escape time* algorithm has been applied to calculate the Mandelbrot set.

Using RISC-pb^{2}l building block notations, the application has been designed by the FastFlow developer as follows:

$scatter_{1 \lhd n} \,.\, [\text{«mandeleq»}]_n \,.\, (gatherall\&drow_{n \rhd 1})$

Appendix D.2 represents the generation of the Mandelbrot application through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 6.2 represents the architectural view of the application in FastFlow. Implemented by the FastFlow developer team, a farm pattern represents the Mandelbrot application.

An instance of `ff_loadbalancer` with a *scatter* as a filter function is used to divide a given plot area evenly among workers.

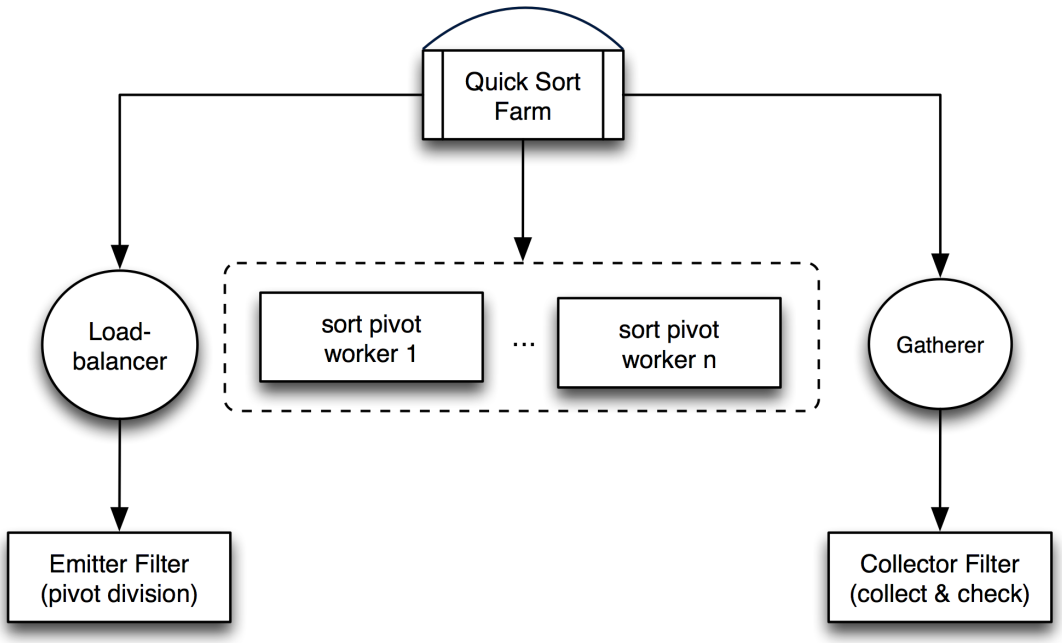


Figure 6.3: The structural composition of components for the quick sort application in FastFlow visualised by structural tree notation

Each worker applies equation 6.1 to the allocated x, y points in the plot area for a certain number of iterations to determine if they have reached a critical *escape* condition. If the condition is reached, the calculation is stopped, the pixel is drawn, and the next x, y points is examined. Otherwise, the point is within the Mandelbrot set.

An instance of `ff_gatherer` with a custom function is used to plot each collected pixel, and where based on the behaviour of that calculation, a colour is chosen for that pixel.

The workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method.

6.1.3 Quick Sort

The quick sort algorithm first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays. [105]

Using RISC-pb^{2}l building block notations, the application has been designed by the FastFlow developer as follows:

$$\overleftarrow{\left(\left(pivot_{dev\,1 \lhd n}\right) \centerdot \left[\text{«sort\_pivot »}\right]_n \centerdot \left(collect\&check_{n \rhd 1}\right)\right)_{termination}}$$

Appendix D.3 represents the generation of the quick sort application through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 6.3 represents the architectural view of the application in FastFlow. Implemented

by the FastFlow developer, a farm pattern represents the quick sort application.

An instance of `ff_loadbalancer` with an custom function is used as a filter. It selects an element, called a pivot, from the array; divides an array into two sub-arrays using the pivot; and sends it to the workers.

Each worker receives a task containing the sub-array and the pivot value for that array. It reorders the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this, the pivot is in its final position.

An instance of `ff_gatherer` with a condition function as a filter is applied. The received sub-array will be assessed by the function to determine whether or not it satisfies the termination condition. If unsuccessful, it returns the sub-array to the emitter for further division.

The workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method. The farm is also equipped with the `ff_farm::wrap_around()` method to implement the feedback.

6.1.4 Fibonacci

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2} \tag{6.2}$$

where $F_0 = 1$ and $F_1 = 1$ [106].

Using RISC-pb^{2}l building block notations, the application has been designed by the FastFlow developer as follows:

$((thresholdBreaker_{1 \lhd n}) \,.\, [\text{«recursiveFib »}]_n \,.\, (\oplus_{n \rhd 1})$

Appendix D.4 represents the generation of the Fibonacci application through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 6.4 represents the architectural view of the application in FastFlow. As it is implemented by the FastFlow developer, a farm pattern represents the Fibonacci application. An instance of `ff_loadbalancer` with a filter function called `thresholdBreaker` is applied to generate the tasks. Based on the provided threshold k, `thresholdBreaker` divides F_n into two sub-parts F_{n-1}, F_{n-2}. For each sub part F_{m_i} the function is called recursively until $mi <= k$. When $m_i <= k$, the F_{m_i} is dispatched to the workers.

Each worker call a recursive function to compute the F_{m_i} recursively.

An instance of `ff_gatherer` with a filter function called reduction is called to reduce the received value of F_{m_i} to an scaler number representing the value of F_n.

The workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()`

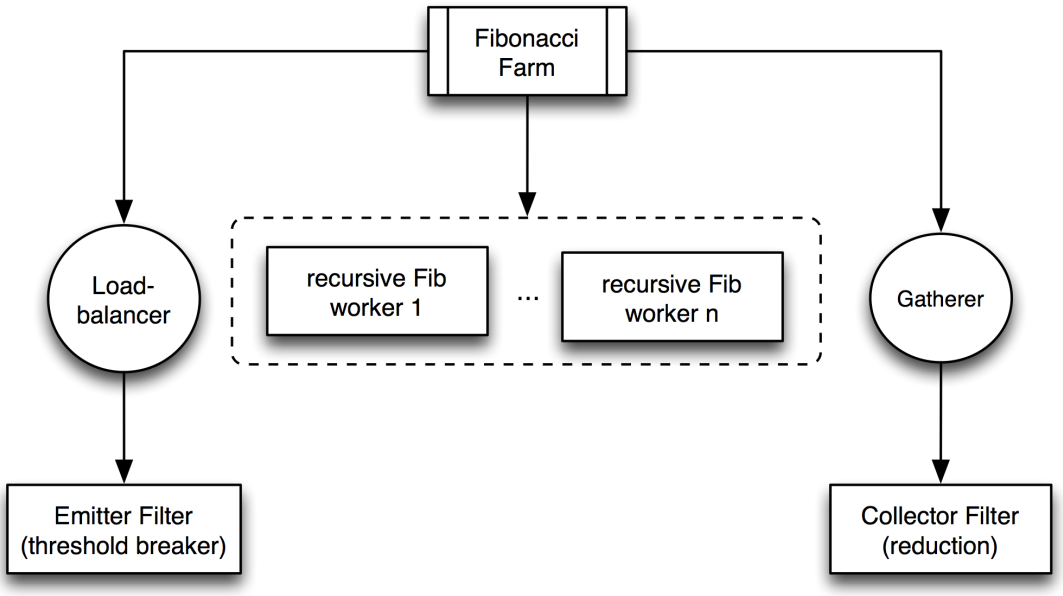


Figure 6.4: The structural composition of components for the Fibonacci application in FastFlow visualised by structural tree notation

method. The farm is also equipped with the `ff_farm::wrap_around()` method to implement the feedback.

6.1.5 Stencil

Stencil computation is a class of algorithms at the heart of most calculations that involve structured (rectangular) grids, including both implicit and explicit partial differential equation (PDE) solvers [107].

It can be defined as a 5-tuple (I, S, S_0, s, T) with the following meaning:

- I = $\prod_{i=1}^{k}[0,\ldots,n_i]$ is the index set. It defines the topology of the array. S is the (not necessarily finite) set of states, one of which each cell may take on any given time-step.
- $S_0 \colon \mathbb{Z}^k \to S$ defines the initial state of the system at time 0.
- $s \in \prod_{i=1}^{l}\mathbb{Z}^k$ is the stencil itself and describes the actual shape of the neighbourhood. There are l elements in the stencil.
- $T \colon S^l \to S$ is the transition function which is used to determine a cell's new state, depending on its neighbours.

Equation 6.3 represents a formal definition of a two dimensional Jacobi iteration [108], implemented on FastFlow.

In this case we start with an initial solution of 0. The left and right boundaries are fixed at 1, while the upper and lower boundaries are set to 0. The number of iterations and the size of the input array are defined by the user.

$$
\begin{aligned}
I &= [0,\ldots,size]^2 \\
S &= \mathbb{R} \\
S_0 &: \mathbb{Z}^2 \to \mathbb{R} \\
S_0((x,y)) &= \begin{cases} 1, & x < 0 \\ 0, & 0 \le x < size \\ 1, & x \ge size \end{cases} \\
s &= ((0,0),(0,-1),(-1,0),(1,0),(0,1)) \\
T &: \mathbb{R}^5 \to \mathbb{R} \\
T((x_0x_1,x_2,x_3,x_4)) &= 0.2 \cdot (\sin(x_0) + \sin(x_1) + \sin(x_2) + \sin(x_3) + \sin(x_4))
\end{aligned}
\tag{6.3}
$$

Using RISC-pb^{2}l building block notations, the application has been designed by the FastFlow developer as follows:

$$\overleftarrow{((customscatter_{1 \lhd n}) \,.\, [\text{«T»}]_n \,.\, (swap \& \oplus_{n \rhd 1}))}_{iteration}$$

Appendix D.5 represents the generation of the stencil application through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 6.5 represents the architectural view of the stencil application in FastFlow. As it is implemented by the FastFlow developer, a farm pattern represents the stencil application.

An instance of `ff_loadbalancer` with a custom scatter function as a filter is applied to distribute the different rows of an input matrix among workers. Each worker applies *(i)* the function T in equation 6.3 to each cell of an allocated row; *(ii)* reduces the row to a scholar number; and *(iii)* returns the number.

An instance of `ff_gatherer` with a filter function has been applied to *(i)* reduce all received values; *(ii)* replace the input with output matrices; and *(iii)* return them to the emitter to apply the next iteration. It also checks the number of iterations for the termination condition.

The workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method. The farm method is also equipped with `ff_farm::wrap_around()` to implement the feedback.

6.1.6 N-queens

The N-queens problem is a generalization of the well-known 8-queens problem. N-queens have to be placed on an $N \times N$ board size chess such that no queen can attack the others. The

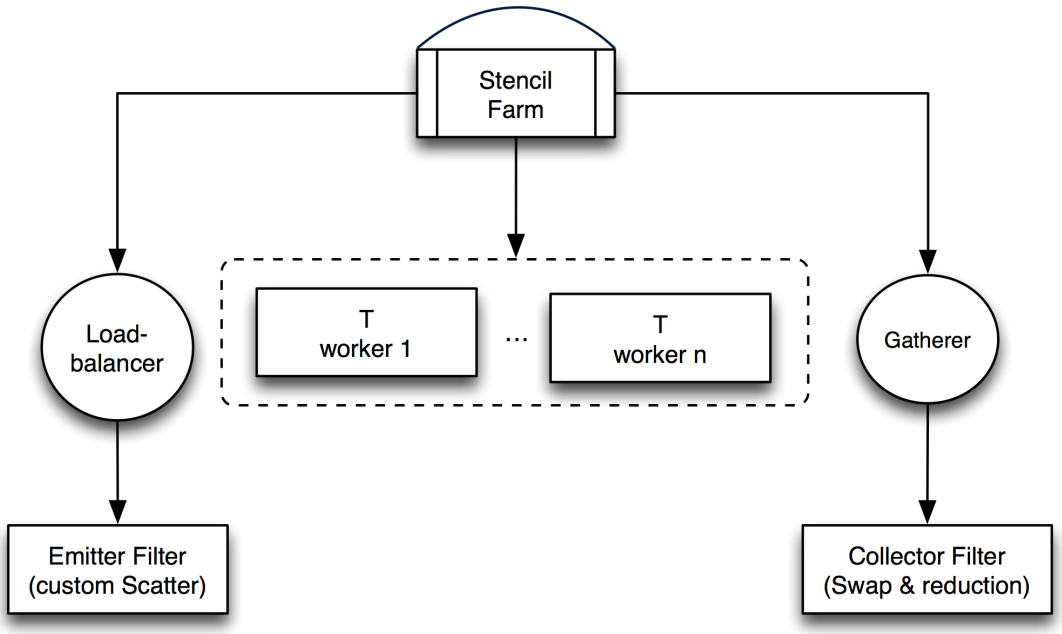


Figure 6.5: The structural composition of components for the stencil application in FastFlow visualised by structural tree notation

objective is to count all possible solutions [109].

One of the fastest sequential implementations of the N-queens problem is Jeff Somer's algorithm [110]. Somer's N-queens is iterative and written in a heavily optimized C code. It finds the number of possible solutions for the N-queens problem as long as the board size does not exceed 21×21.

Representing possible solutions as a decision tree:

- The root of the tree represents 0 choice;
- Nodes at depth 1 represent the choices for the first row in the board;
- Nodes at depth 2 represent the choices for the second row in the board;
- A path from the root to a leaf represents a candidate solution for n-queen.

Using RISC-pb^{2}l building block notations, the application has been designed by the FastFlow developer as follows:

$(CustomScatter_{1 \lhd n}) \,.\, [\text{«SommersNqueen»}]_n$

Appendix D.6 represents the generation of the *N-queen* application through the RISC-pb^{2}l building block presented in Listing 3.1.

Figure 6.6 represents the architectural view of the *N-queen* application in FastFlow. As it is implemented by the FastFlow developer, a farm pattern represents the N-queens application.

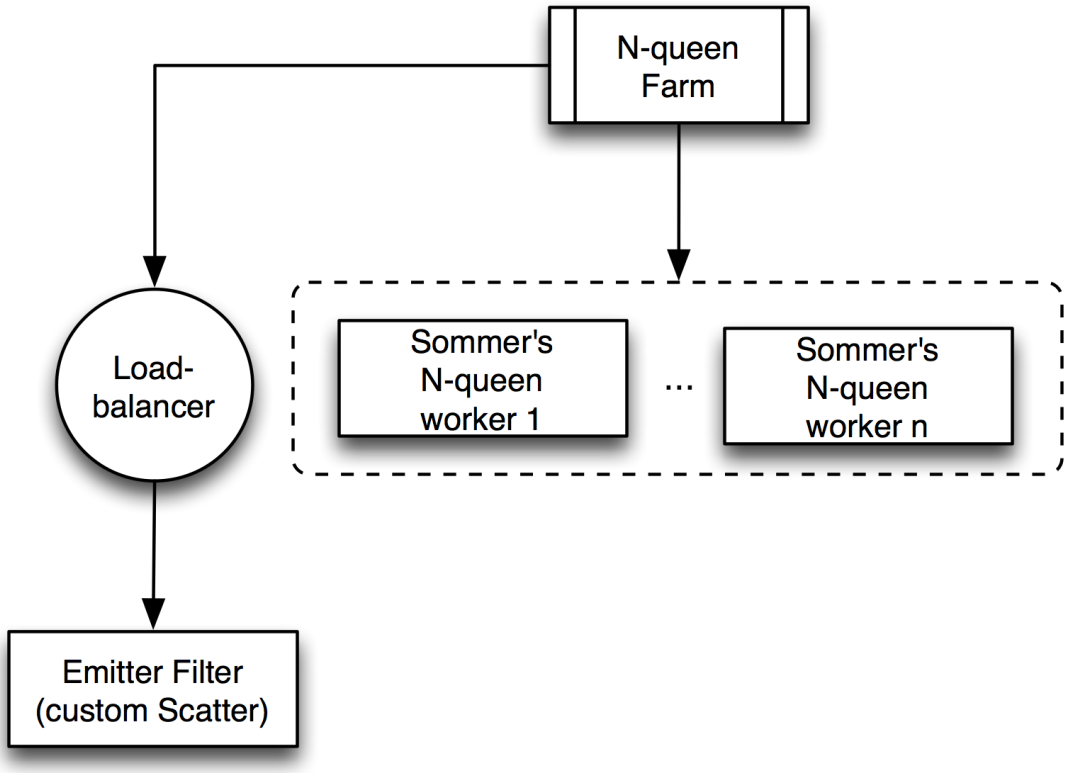


Figure 6.6: The structural composition of components for the *N-queen* application in FastFlow visualised by structural tree notation

An instance of `ff_loadbalancer` with a custom filter function has been applied. The `CustomScatter` function adjusts the board until depth K in the tree and then sends it to the workers to complete it.

Each worker determines all possible solutions for the given sub-tree. K can be determined by a user.

The workers and emitter are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method.

6.2 Heterogeneous Applications

This section contains three different existing FastFlow applications that use *ParWrapper* to encapsulate the existing OpenMP based or GPU based blocks of a programme inside FastFlow. By applying these applications we demonstrate the efficiency of PEI for executing applications that use the GPU device but do not use the proposed *HWrapper* (The OpenCL back-end) on heterogeneous multi-core architectures.

Table 6.2 represents the characteristics of each application. The first column of the table represents the application name and the second column represents the features that the application covers.

6.2.1 Custom Implementation of EISPACK Routines

The validation of the custom implementation of the EISPACK routines for solving Hermitian eigensystems (originally introduced in [111]) has been considered as an existing practical problem.

The application setting bears an architectural similarity to the final intended deployment and presents a similar computational demand.

The EISPACK application has three parts:

1. **Generation** of suitable Hermitian test matrices, A.

2. **Solution** using the test GPU kernels to compute the eigenvectors E and eigenvalues D in the GPU.

3. **Verification** of the computed eigenvectors and eigenvalues in the CPU. For a matrix of eigenvectors E and a corresponding diagonal matrix of eigenvalues D, we expect that $AE = ED$. Therefore, a possible error function is the Frobenius norm of the matrix

$$\epsilon = ||AE - ED||_F$$

Table 6.2: Summary of Heterogeneous Applications Characteristics.

Application Name	Application Characteristics	Input Features
EISPACK Routines	We have used the FastFlow pipeline pattern to augment the palletisation of EISPACK routines presented in [111]. This application is a demonstration of embedding FastFlow PEI inside a distributed application on Xookik cluster. It is distributed by MPI on 4 symmetric nodes of Xookik cluster and in each node a 3-stage FastFlow pipeline pattern augments the level of parallelism. The second stage of Pipeline for each node is an instance of *ParWrapper* component. The component is implemented in CUDA and can only run on GPU. However, the first and last stages of the Pipeline pattern are instances of *ParWrapper* component that embed OpenMP based code. These two stages can only run on CPU. As a large-scale heavy workload application, EISPACK routines benefits from dynamic memory management implemented in PEI to prevent queue overflows for Pipeline components.	No Matrices = 55296 pseudo-random Hermitian matrices of order 1024 and double precision.
SMTWTP	Defined in [112], this application is a reduction composition component (i.e. Farm with collector in FastFlow) with Feedback that has two workers. The first worker is a reduction composition component (i.e. Farm with collector in FastFlow) with n instances of *ParWrapper* component as workers. Each *ParWrapper* component is implemented in CUDA and can only run on GPU. The second worker is a Farm pattern with m instances of *SeqWrapper* component as workers. All *SeqWrapper* components are implemented in C++ and can only run on CPU. m and n are two determining factors that can affect the application performance. Determining the optimal values of n and m depends on the underlying architectures. The static structural configuration in PEI is used to tune n and m for Titanic machine.	No of Jobs=100
MD	Defined in [112], this application is a non-reduction composition component (i.e. Farm without collector in FastFlow) with two workers. The first worker is a reduction composition component (i.e. Farm with collector in FastFlow) with n instances of *ParWrapper* component as workers. Each *ParWrapper* component is implemented in CUDA and can only run on GPU. The second worker is a non-reduction Farm pattern (Farm without collector) with m instances of *SeqWrapper* component as workers. All *SeqWrapper* components are implemented in C++ and can only run on CPU. m and n are two determining factors that can affect the application performance. Determining the optimal value of n and m depends on the underlying architectures. The static structural configuration in PEI is used to tune n and m for Titanic machine.	No of Molecules= 1000.

where the Frobenius norm of an $m \times n$ matrix M is defined as

$$||M||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|m_{ij}|^2}$$

.

Using RISC-pb^{2}l building block notations, the FastFlow pattern embedded in the EISPACK application has been designed as follows [93]:

‹|Generation|› .‹|Solution|› .‹|Verification|›

Using the RISC-pb^{2}l building block presented in Listing 3.1, Appendix D.7, represents the generation of the FastFlow farm pattern encapsulated in the EISPACK application.

Figure 6.7 illustrates an architectural view of the EISPACK application that encapsulates the FastFlow pipeline pattern. The FastFlow implementation of the application is provided in [93].

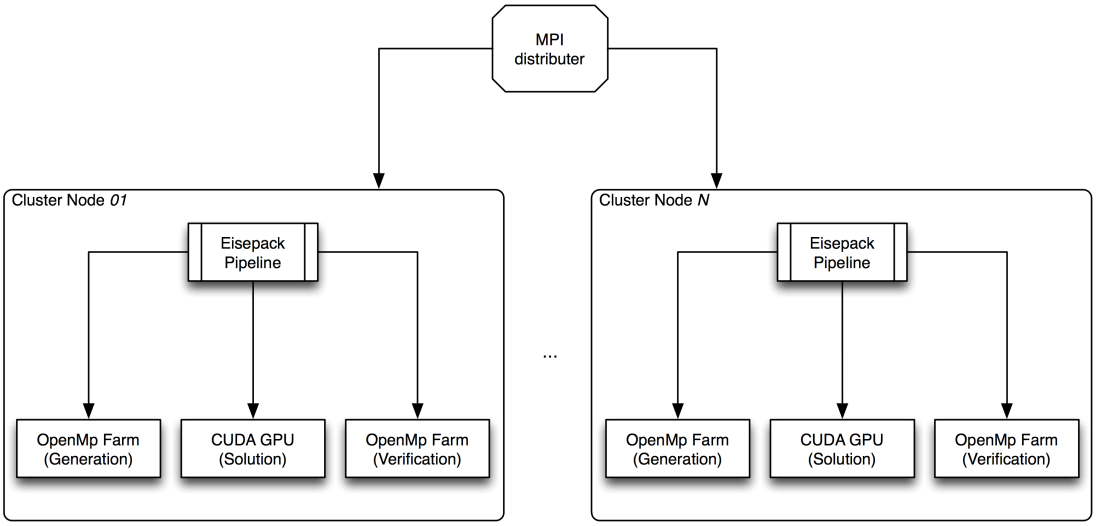


Figure 6.7: The structural composition of components for the EISPACK application encapsulating the FastFlow pipeline patter. It is visualised by structural tree notation. The application is composed of a three-level hierarchy of skeletons, consisting of a distributed farm of MPI workers at the top-level that contains nested pipeline workers with nested OpenMP farm stages and a GPU stage.

In FastFlow the EISPACK application can be demonstrated as a 3-stage pipeline.

The pipeline stages exchange batches of work based on the optimal requirements for the GPU solution stage where the performance is reliant on the CUDA grid size at kernel launch.

Generation and verification in the first and final stages of the pipeline operate on these individual batches of independent test problems. This presents another level of parallelism that is well suited to the farm skeleton. Employing skeletal composition allows a nested parallel hierarchy.

The `Generate_stage`, `Solve_stage` and `Verify_stage` represent the initial, intermediate and terminal stages of the pipeline.

The `Solve_stage` invokes our external GPU-enabled linear algebra libraries. The application implementation uses OpenMP inside **Generation** and **Verification** function to implement the farm-worker pattern.

The pipeline stages are added to a `ff_pipeline` container object in the proper sequence. Execution is started by an invocation of the `ff_pipeline::run_and_wait_end()` method.

The application already has an additional level of nesting that uses an MPI farm to handle distribution over multiple nodes in a cluster. This will also demonstrate that FastFlow can be embedded as part of an application to boost the throughput.

6.2.2 SMTWTP

The Single Machine Total Weighted Tardiness Problem (*SMTWTP*) is defined as follows [112]. Given n jobs and each job, i, is characterised by its processing time, p_i, deadline, d_i, and weight, w_i, the goal is to schedule the execution of jobs in a way that achieves the minimal total weighted *tardiness*. The tardiness of a job is defined by $T_I = \max\{0, C_i - d_i\}$ (with C_i being the completion time of the job, i) and the total tardiness of the schedule is defined as $\sum w_i T_i$.

An ant colony optimisation solution [113] is applied to the SMTWTP problem which consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail*. The successful routes are determined by stronger pheromone trails which are defined by a matrix, τ, where $\tau[i, j]$ is the preference of assigning job j to the i-th place in the schedule. After all the ants have computed their solution, the best solution is chosen as the 'running best'; the pheromone trail is updated accordingly, and the next iteration is started. The algorithm is composed of three parts:

- **Find Solution**: Finds the solutions for all ants.
- **Best Solution**: Choose the best solution.
- **Pheromone Update**: By considering the current best solution this updates the pheromone trail.

An optimal pattern combination for developing this application is provided by a filtering method that represents the only parallelisable part is the getSolution component [112].

Using RISC-pb^{2}l building block notations, the FastFlow Farm pattern for the getSolution component embedded in the *SMTWTP* application has been designed as follows [93]:

$$\overleftarrow{(unicast_{1\triangleleft 2} \,.\, [scatter_{1\triangleleft n} \,.\, [\langle\!\langle findSol_{CPU} \rangle\!\rangle]_n \,.\, gatherall_{n\triangleright 1}\,,}$$
$$\overrightarrow{unicast_{1\triangleleft m} \,.\, [\langle\,|\; findSol_{GPU} \;|\,\rangle]_m \,.\, gather_{m\triangleright 1}] \,.\, (gAll\&g_{best}\&u_{pheromone_{2\triangleright 1}}))_{iter}}$$

Using the RISC-pb^{2}l building block presented in Listing 3.1, Appendix D.8 represents the generation of the FastFlow farm encapsulated in the Eispack application.

Figure 6.8 illustrates an architectural view of the *SMTWTP* application encapsulating the FastFlow farm pattern. The FastFlow implementation of the application is provided in [112].

In FastFlow the *SMTWTP* application can be demonstrated as a farm pattern with two nested farms as workers.

An instance of `ff_loadbalancer` with a *unicast* filter is applied to distribute different tasks to different workers to find the solutions.

Each worker operates on its allocated tasks to generate the solution.

For CPU_{sol}, a farm pattern with a *scatter* is applied to further divide a task among its workers. An instance of `ff_gatherer` with a *gatherall* filter is applied to collect and assemble the divided tasks of the nested Farm.

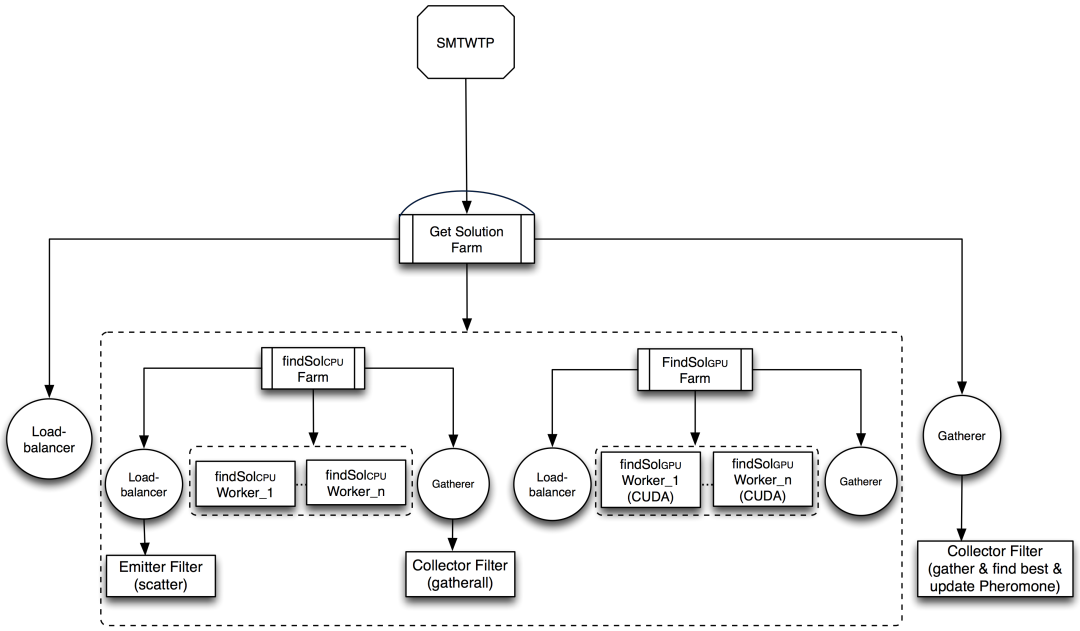


Figure 6.8: The structural composition of components for the *SMTWTP* application in FastFlow visualised by structural tree notation

For GPU_{sol}, a farm pattern with a *unicast* is applied to delegate the received tasks to one of the GPU workers. Also, an instance of `ff_gatherer` with a simple *gather* filter is applied to collect the tasks and pass them to the outer `ff_gatherer`.

For the outer farm, an instance of `ff_gatherer` with a custom filter function is applied which contains a *gatherall* filter with a barrier to receive all the solutions. Once all the tasks have been received, it applies the g_{best} and $u_{pheromone}$ methods to find the best solution and updates the pheromones.

For each farm pattern, the workers, emitter and collector are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method.

The outer farm is also equipped with the `ff_farm::wrap_around()` method to implement the feedback.

6.2.3 Molecular Dynamics

The molecular dynamics (MD) simulation computes a system of N particles at the atomic level [114]. Once the system has been initialised, the interactions between the molecules are evaluated explicitly, allowing for the numerical integration of Newton's equations of motion. The molecular trajectories in time yield the thermodynamic properties of the system.

The molecular simulation code used here (CMD) is designed for conducting basic research into HPC MD. In the BasicN2 variant investigated here, all intermolecular distances are evalu-

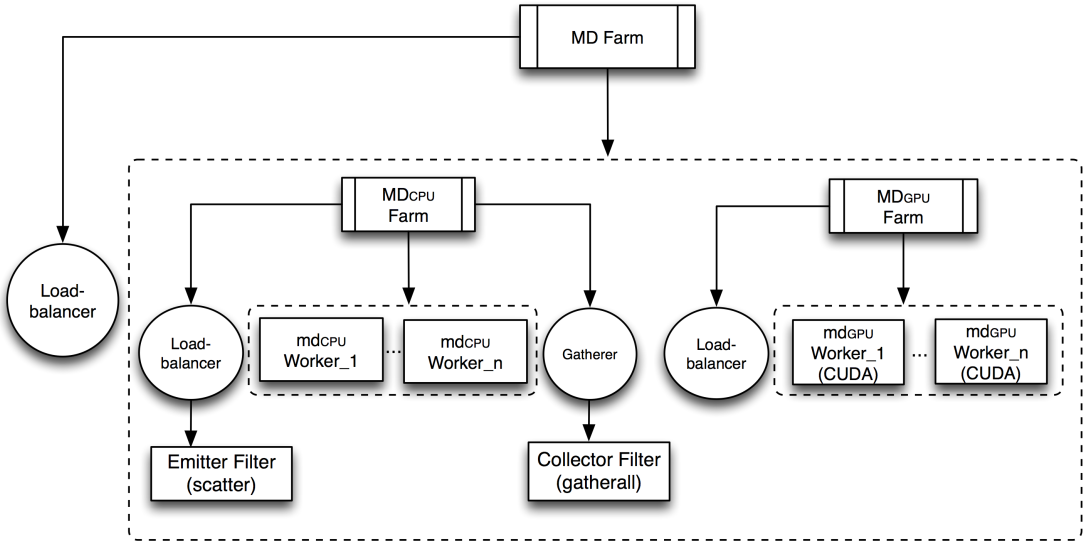


Figure 6.9: The structural composition of components for the *MD* application in FastFlow visualised by structural tree notation

ated in order to identify the interaction partners. However, a special flavour of BasicN2 is used, where the domain is decomposed into sub-domains of approximately 1000 molecules in order to counter the prohibitive scaling of the neighbour's search (otherwise $O(N^2)$).

As inferred from the profiling data, the force calculation routine dominates the simulation time and is therefore parallelised. The force calculation itself is decomposed into two kernels, intra-domain and inter-domain (with the use of halos) interactions.

To execute both intra-domain and inter-domain interactions, a filtering method determines that both intra-domain and inter-domain interactions should be merged with each other into one unit of computing function and executed sequentially for each sub-domain. However, each sub-domain can be executed in parallel [112].

Using RISC-pb^{2}l building block notations, the *MD* application has been designed as follows [93]:

$$unicast_{1 \lhd 2} \centerdot [scatter_{1 \lhd n} \centerdot [\langle\langle md_{cpu} \rangle\rangle]_n \centerdot gatherall_{n \rhd 1} \ , \ unicast_{1 \lhd m} \centerdot [\langle \mid \ md_{gpu} \ \mid \rangle]_m]$$

Using the RISC-pb^{2}l building block presented in Listing 3.1, Appendix D.9 represents the generation of the *MD* application.

Figure 6.9 illustrates an architectural view of the *MD* application in FastFlow. The FastFlow implementation of the application is provided in [112].

In FastFlow, the *MD* application can be demonstrated as a farm pattern with two nested farms as workers.

An instance of `ff_loadbalancer` with a *unicast* filter is applied to distribute different tasks to different workers to find the solutions.

Each worker operates on its allocated tasks to generate the solution.

Device	Intel Westmere E5620
Number of Cores	4
Clock Speed	2.4 GHz
Single Precision Floating Point Performance (perCore)	1.51 GFLOPs
Intel® Smart Cache	12 MB
Instruction Set	64-bit
Memory Size	24GB
Max Memory Bandwidth	25.6 GB/s

Table 6.3: Intel Westmere E5620 quad core processor for one of the worker nodes in Eddie. Eddie is the compute component of Edinburgh Compute and Data Facility (ECDF), which is located at the advanced computing facility of the University of Edinburgh

For CPU_{sol}, a farm pattern with a *scatter* is applied to further divide a task among its workers. An instance of `ff_gatherer` with a *gatherall* filter is applied to collect and assemble the divided tasks of the nested farm.

For GPU_{sol}, a farm pattern with a *unicast* filter is applied to delegate the received tasks to one of the GPU workers.

For each farm pattern, the workers and emitter are added to a `ff_farm` container object in the proper sequence. Execution is started by an invocation of the `ff_farm::run_and_wait_end()` method.

6.3 Application Evaluation

The evaluation of PEI for the existing applications has been carried out on three platforms. Tables 5.3 and 6.3 represent the hardware specifications of the applied test machines respectively.

6.3.1 Performance Overhead

Two types of overhead evaluations have been considered. The first type of evaluation indicates the overhead of the building block based approach on application scalability with highly specialised skeleton frameworks. FastFlow, SKePU and Thrust have been considered as candidate frameworks for this evaluation, where Fastflow represents a building block based framework and SKePU and Thrust represent the highly specialised skeleton frameworks.

The second type of evaluation states the overhead of PEI by determining the extra execution time of the instrumentations added to FastFlow.

Software	Version
gcc	4.1.2-50.el5
Red Hat Enterprise Linux Server release 5.4 (Tikanga)	2.6.18-238.1.1.el5
FastFlow	1.1.0
SkePU	0.6
Thrust	1.5.0

Table 6.4: List of the software used to evaluate the framework overhead on the Scalability of applications

6.3.1.1 Framework Overhead on the Scalability of Applications

We have studied the gravitational force of the *N*-body simulation for 20 step iterations in FastFlow, Thrust and SKePU. Table 6.4 represents the software specification applied in this experiment. This experiment has been carried out on one of the worker nodes in Eddie which is represented in Table 6.3.

A detailed implementation of the computation functions for each framework has been reported in Appendix E.

Figure 6.10, represents the execution time of the *N*-body problem for 1024 bodies implemented in the three different frameworks. The x-Axis represents the number of cores that has been used for an application execution. The baseline for speed-up is the optimised sequential version provided by the SICSA multi-core challenge [103].

As stated in the figure, the scaling is not linear. Using the sensor information provided for the building block approach, Table 6.5 represents the execution times of each component in FastFlow computing for the gravitational force in the *N*-body problem. The runtime for computing the gravitational force for each worker is almost reduced by half as the number of cores is doubled, i.e., exhibits linear scalability. The runtime for the emitter and the collector filter is almost negligible in comparison to the workers' runtime. Running the sensor information provided for FastFlow reveals that for 1024 bodies the overhead of running the farm is 0.0035 seconds.

Therefore, computing the gravitational force for 20 step iterations will generate 0.07 seconds overhead in the system which is unavoidable. This overhead time is not negligible in comparison to the workers' runtime and is the reason for the non-linear scaling with 1024 bodies. The overhead time is spent to push/pop the tasks into/from workers and the collector queue. By increasing the size of the problem, the parallelism overhead becomes negligible. As shown in Figures 6.11 and 6.12, by increasing the number of cores for 8192 and 65536 the algorithm runtime is reduced by 50% and the result is one order of magnitude faster than the

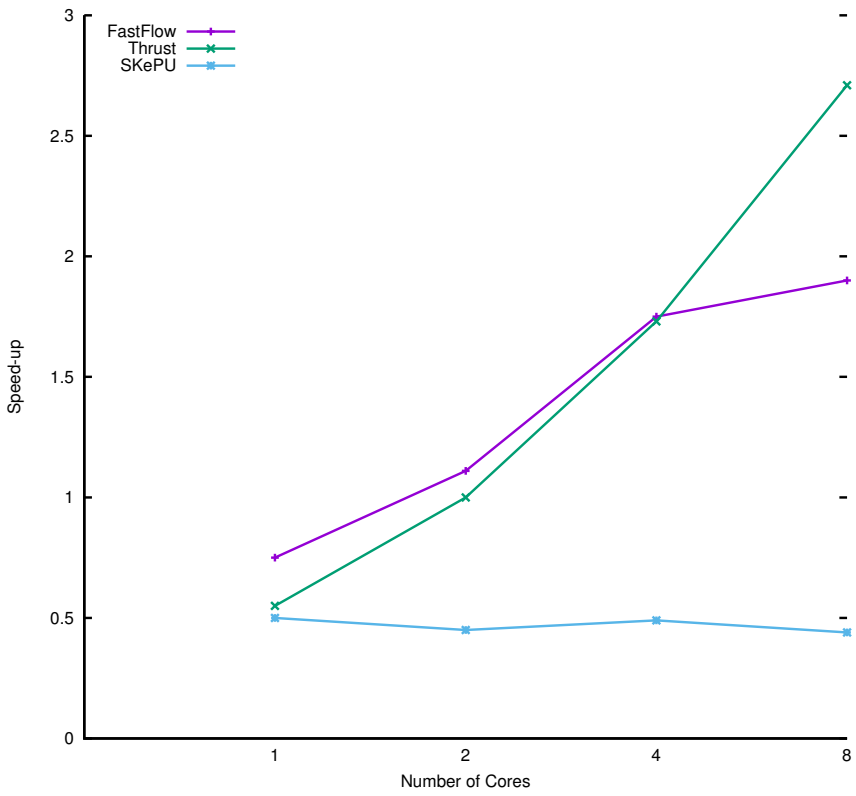


Figure 6.10: Speed-up graph for N-body simulation using FastFlow, Thrust and SkePU (Problem size: 1024 bodies)

No. of core	Emitter time	Worker time	Collector time	Total
1	0.000023	0.272687	0.001204	0.360081
2	0.000034	0.153967	0.001972	0.240158
4	0.000162	0.099170	0.003533	0.155273
8	0.000234	0.059708	0.007017	0.134568

Table 6.5: The detailed execution times of the FastFlow farm for calculating the gravitational force on the CPU for 1024 bodies

serial algorithm.

As per our evaluation figures plotted in Figure 6.10, SKePU does not seem to scale well for 1024 bodies. Furthermore, the gravitational force of the *N*-body simulation in SKePU is composed of three macro functions which has been explained in Appendix E. As reported in Table 6.6, the only macro which scales by increasing the number of cores is the magnitude function while the other macros have a worse runtime than when running on one core. The magnitude function is only run once, while the other two run 6 times per each iteration. The runtime of these two functions together is more than that of the magnitude. Hence, even the reduction in magnitude runtime caused by increasing the cores is less than the increase in the others' runtime and as such it does not scale.

The reason for this is that these two functions only perform one or two basic instructions over the CPU. When they run over one core, the maximum number of blocks of data is fetched from the memory to the cache and the instructions executed on them. For small-size problems, such as 1024 bodies, the overhead of parallelising a function with a single instruction is considerably large, and even on a par with the serial runtime.

When the size of the problem is small, increasing the number of threads to parallelise the difference and update_velocity macros which only run one or two basic instructions, increases the number of memory accesses. The time spent on memory access is therefore more than that for computing a single instruction. Consequently, the algorithm does not perform efficiently.

For 8192 bodies, the runtime for magnitude is the dominant runtime of the algorithm and the algorithm scales almost linearly, as shown in Figure 6.11.

When increasing the size of the input to 65536, the whole bodies vector cannot fit in the cache. Then, the number of memory accesses required to read the data is more than the number of cores. As the memory accesses are overlapped and pipelined, the scaling becomes linear since the cores remain busy as shown in Figure 6.12.

It is therefore clear that when the size of the problem is small, typically under $8,192$ bodies for our study, the memory access times dominate the performance figures. As the problem grows, memory access patterns become less dominant and scalability tends to improve.

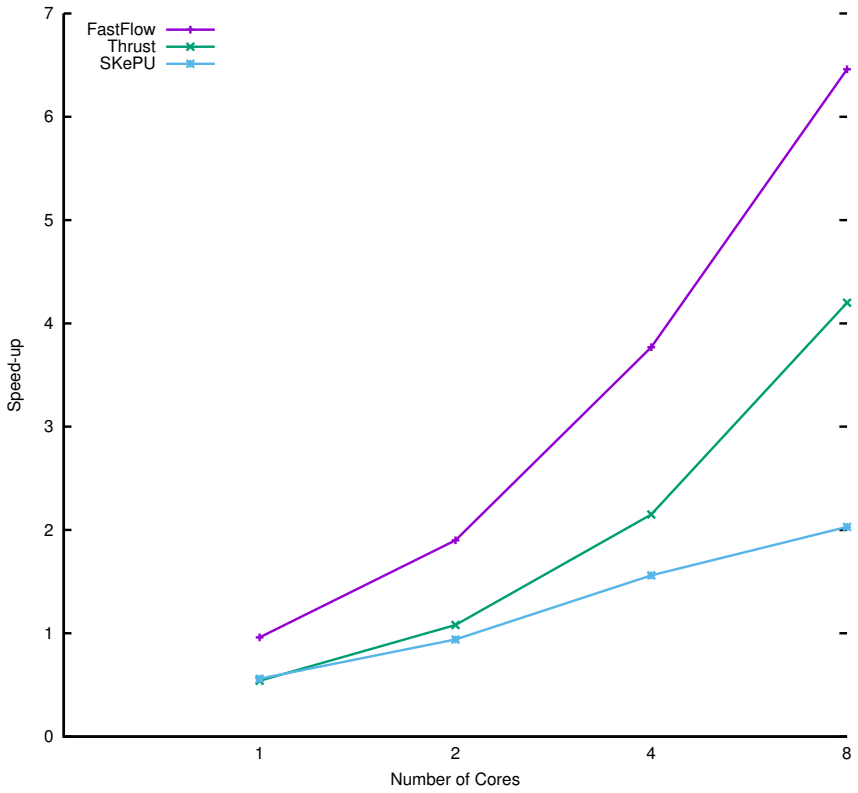


Figure 6.11: Speed-up graph for N-body simulation using FastFlow, Thrust and SkePU (Problem size: 8192 bodies)

No. of core	Difference	Magnitude	Update velocity	Total
1	0.03599	0.27782	0.04632	0.53453
2	0.06226	0.18604	0.07377	0.59862
4	0.06186	0.12588	0.08193	0.56203
8	0.07501	0.11355	0.08801	0.61139

Table 6.6: The detailed execution times of each function for calculating the SKePU gravitational force on the CPU for 1024 bodies

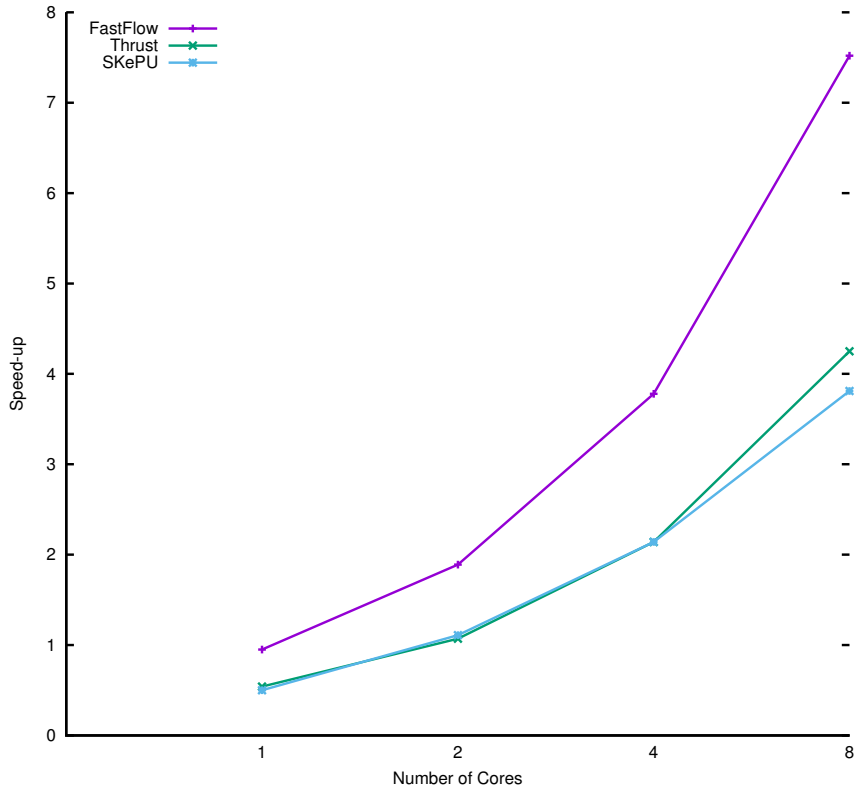


Figure 6.12: Speed-up graph for N-body simulation using FastFlow, Thrust and SkePU (Problem size: 65536 bodies)

As shown in Figures 6.10, 6.11 and 6.12, Thrust scales linearly with the number of cores for all problem sizes. Nevertheless, the careful examination of the performance figures for all algorithms on one core indicates that Thrust has the highest overhead for parallelisation, even more than that of SKePU. However, because it scales linearly, the overall runtime of the system is better than that of SKePU. The following reason distinguishes Thrust from SKePU for scaling well

1. Thrust provides a C++ function object template, called functors, for the user to generate their own function, which gives flexibility to the computation structure.

2. Thrust provides a patten called `make_tuple`, which combines all one dimensional variables, which is similar to an object in C++. This provides more flexibility in memory management.

For each framework the runtime of *N*-body executed on one core can be compared to that of the serial version. Thus, from Figures 6.10, 6.11 and 6.12, it is apparent that FastFlow has the least overhead among all frameworks.

Also, FastFlow scales better than the two other frameworks when the size of the problem is increased. The reason for this is that by taking advantage of the building block approach, FastFlow accepts any arbitrary function with any input data structure format. This will allows the programmer to further optimise the memory access computation functions, when required, for different applications. As in this case, the computation functions in FastFlow have employed the triangular approach with adjustable tiling for the data structure format. This makes FastFlow twice as fast as Thrust and SKePU.

On the one hand, providing more flexibility in memory management and computation functions requires more effort in designing and developing a parallel application and a programmer needs to think in parallel.

On the other hand, by supporting any arbitrary data structure executed by any arbitrary computational function, a wider range of applications can be covered, and specifically even existing applications can be accelerated with minimum modifications.

6.3.1.2 PEI Overhead

Figure 6.13 compares the execution time of the original FastFlow framework and the instrumented version **without** any dynamic coordination decisions (for both aggressive and sparse sampling) on a node of the Xookik cluster.

For the quick sort application where the input stream size is of the order of 10 million, the overhead for the aggressive mode is less than 3%. For most cases, the performance drop is less than 1%.

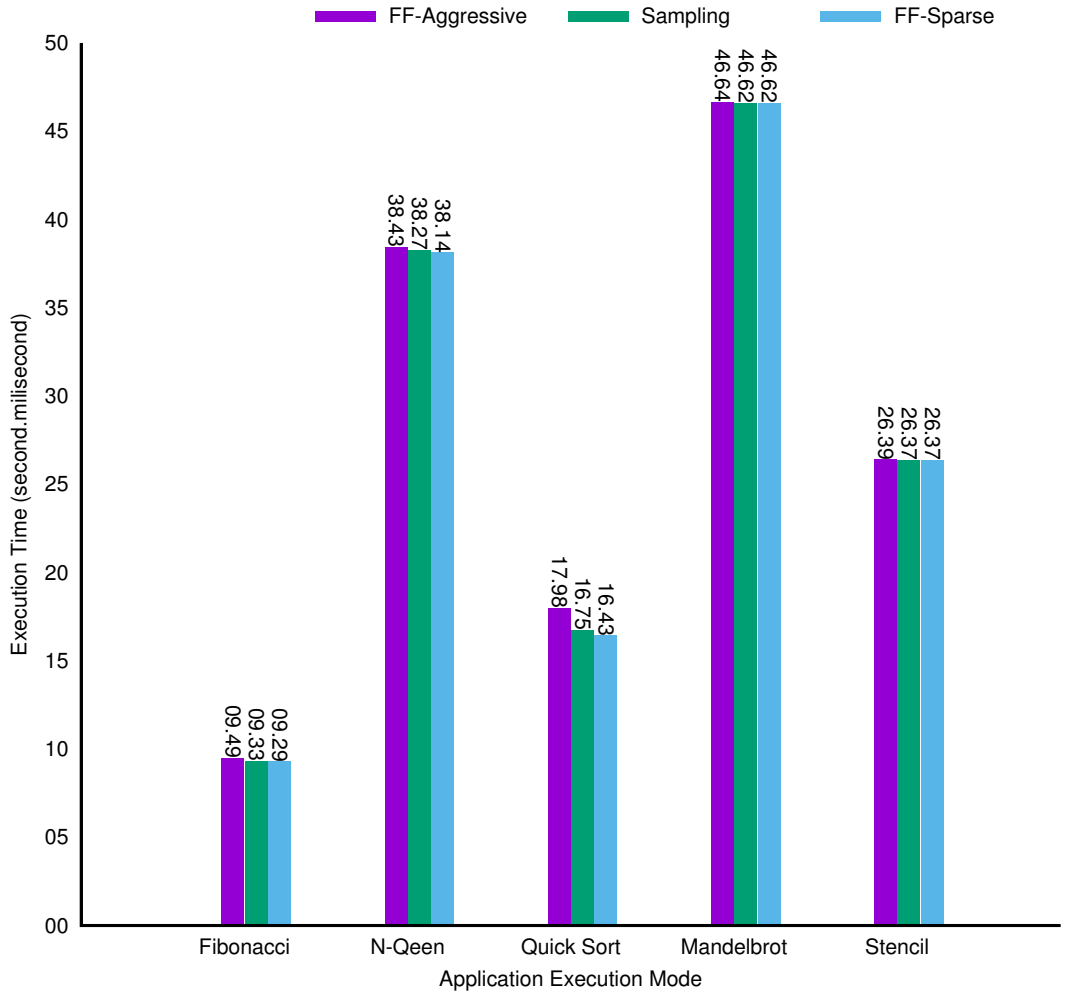


Figure 6.13: The upper-bound overhead of performance metrics tracing over the FastFlow framework on a node of the Xookik cluster for FastFlow benchmark applications. The number of workers for each farm is 8

Table 6.7: Software Specification used for FastFlow benchmark applications

Application name	Input specification
Mandelbrot	canvasSize =1024*1024 iterations=25000
Quick sort	arraySize =int[50000000]
Fibonacci	n=40
N-queen	boardSize =18*18 depth= 4
Stencil	inputMatrix=double[16384][16384] iterations=10

6.3.2 Efficient Idling

Figure 6.14 demonstrates the effect of the efficient idling technique for PEI, offered in Chapter 4, section 4.1.4. The applications' specifications, selected for this experiment, are provided in Table 6.7. The experiment has been carried out on a node of the Xookik cluster.

Each application presented in Table 6.7 has been executed with and without using the efficient idling technique. Considering the fact that a node in the Xookik cluster has 12 cores, two different constructions of farm pattern have been considered for each application. The former construction has 8 workers for each application and the latter construction has 12 workers for each. Knowing that each farm has 2 more component namely the emitter and the collector, the number of components for the first construction is less that the number of available cores and for the second one is more than the number of available cores.

Figures 6.14 demonstrates that increasing the number of workers from 8 (column 2) to 12 (column 4) for FastFlow **without** efficient idling shows a very aggressive drop in performance. This can be up to a 200% drop in speed up when using the busy waiting loop technique. Moreover, increasing the number of workers from 8 (column 1) to 12 (column 3) for FastFlow **with** the efficient idling technique not only prevents a performance drop for the applications but also, for some cases, it can achieve up to a 27% improvement in application runtime.

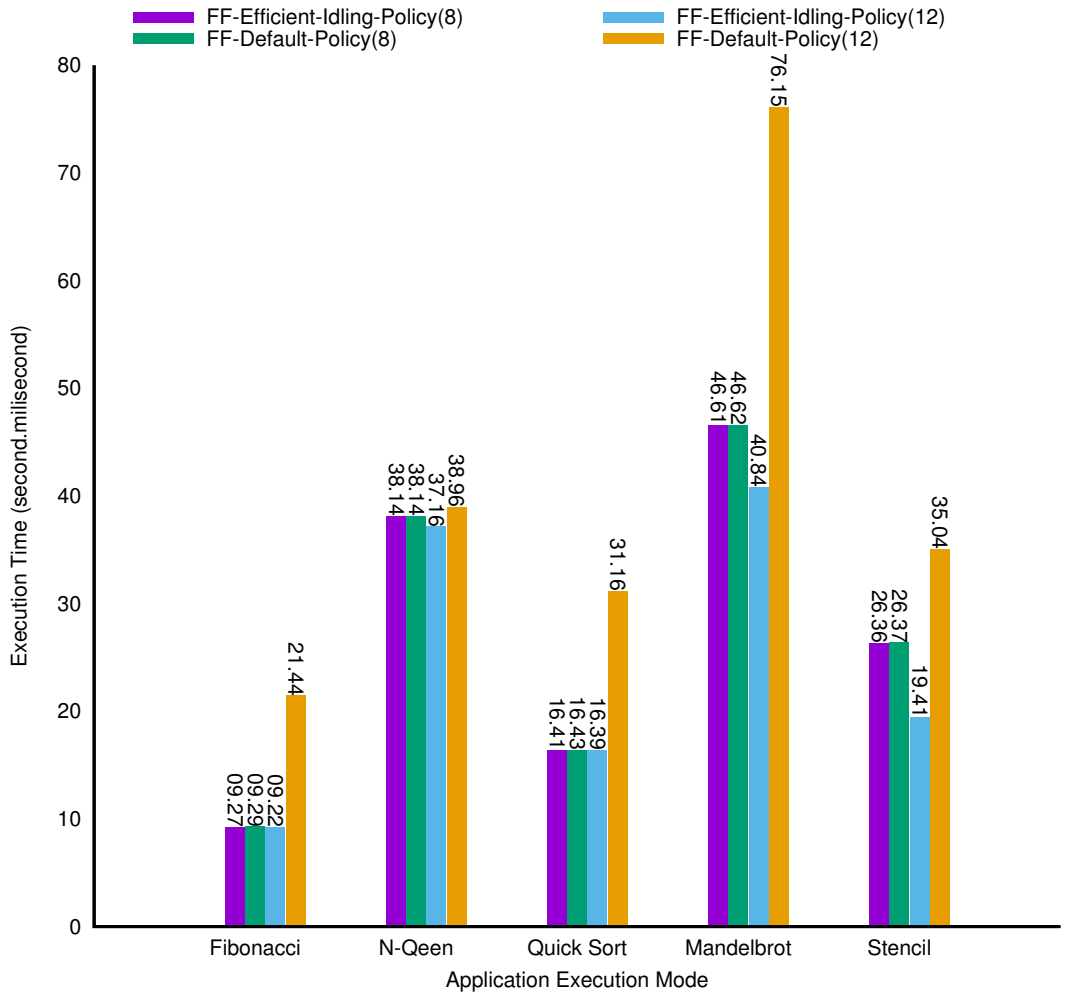


Figure 6.14: The applications' runtime with and without using efficient idling technique

Table 6.8: Active execution times on individual GPUs and total programme runtime for the EISPACK application on all 4 nodes of the Xookik cluster

Resource	Runtime (seconds)
GPU1	8995.6
GPU2	8930.1
GPU3	8930.7
GPU4	8995.4
Total Cluster	9227.0

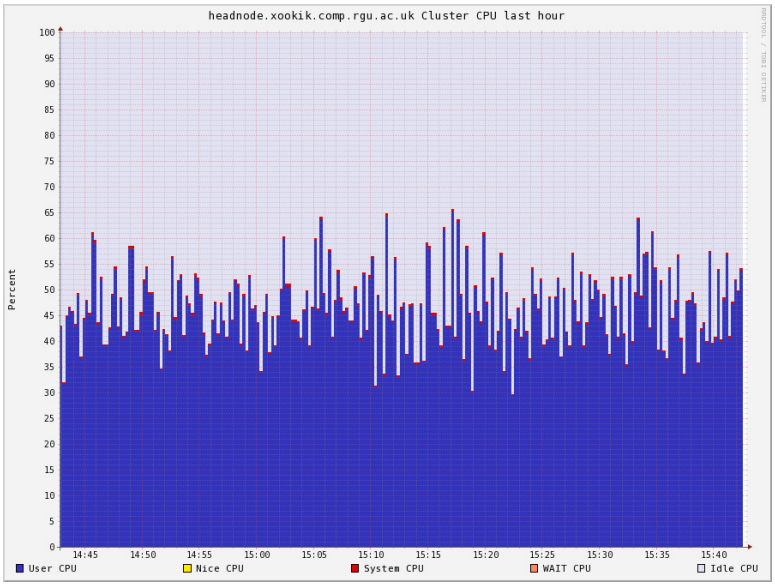


Figure 6.15: The overall cluster CPU usage percentage over one hour of execution for the EISPACK application

6.3.3 Memory Management

An evaluation of the the memory management technique for PEI, offered in Chapter 4, section 4.1.3, was carried out on a 4-node multi-GPU Xookik cluster with 4 MPI processes and 12 workers at each nested farm in the pipeline stage. 55296 pseudo-random Hermitian test matrices of order 1024 and at double precision were streamed through the pipeline to establish that the computed error is within the acceptable tolerance.

Table 6.8 indicates that while the total application runtime in the cluster was 9227 seconds (or 2.53 hours), all GPUs in the cluster were computationally active for a minimum of 8930 seconds (or 2.48 hours). The close correspondence between the active GPU computing times and the total application runtime is an indicator of high GPU utilisation with minimal idling.

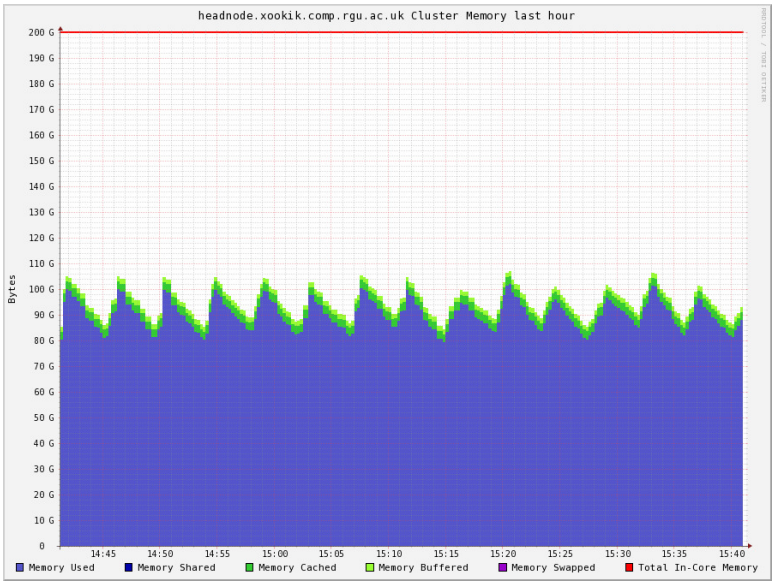


Figure 6.16: Overall memory usage over one hour of execution the EISPACK application on all four nodes of the Xookik cluster

Therefore, as expected, the GPU stage constitutes the primary bottleneck to the pipeline. Figure 6.15 indicates that CPU usage varies between 30% and 60% following variations in the number of active processes. As the GPU stage constitutes the bottleneck, throttling adaptively imposes a limit on CPU usage, preventing pipeline queue overflows.

Adaptively controlling the memory usage is crucial here to allow FastFlow to scale up to large problems. These measures are also suited to machines with a constrained main memory, as was the case with our original development platform.

Figure 6.16 presents memory usage over one hour of execution for the entire cluster. The proposed adaptive memory usage control throttles the pipeline configuration to further manage the allocated memory. The following STOP_THRESHOLD and START_THRESHOLD are respectively used to control the memory between 50% and 40% of the total 200 GB available physical memory. The distinctive saw-tooth waveform follows form intermittent throttling of the pipeline.

6.3.4 Static Structural Configuration

An evaluation of the static structural configuration for PEI, offered in Chapter 4, section 4.4.4, has been carried out on the SMTWP and MD applications over Titanic [112].

SMTWTP Using the reduction rules provided in equation 4.12, the SMTWTP abstract computation graph is:

$\wedge(\langle\langle findSol_{CPU} \rangle\rangle \vee \langle\langle\ findSol_{GPU}\ \rangle\rangle)\wedge.$

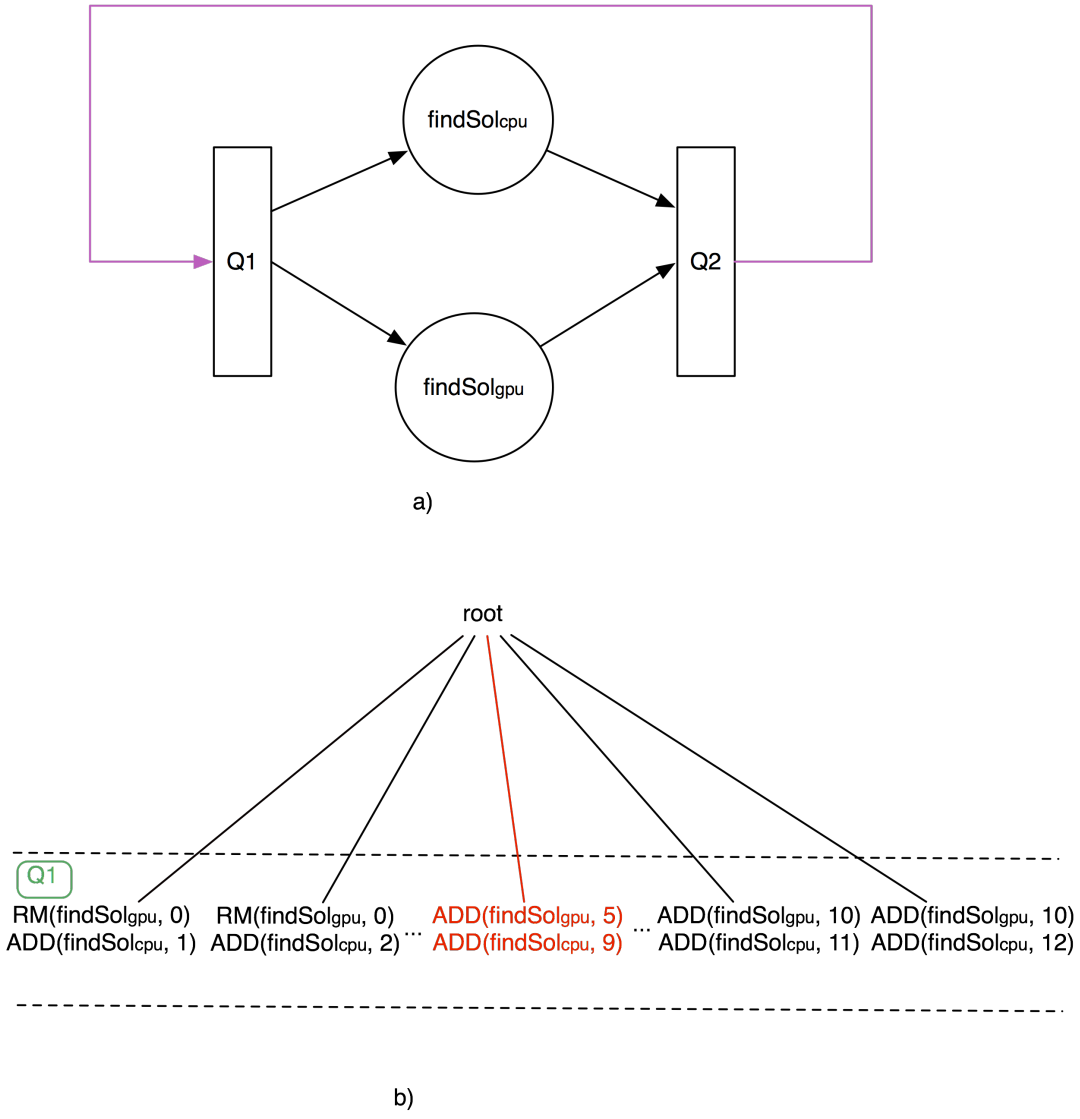


Figure 6.17: a) Abstract computation graph for SMTWTP. b) MCTS decision tree generated for graph a

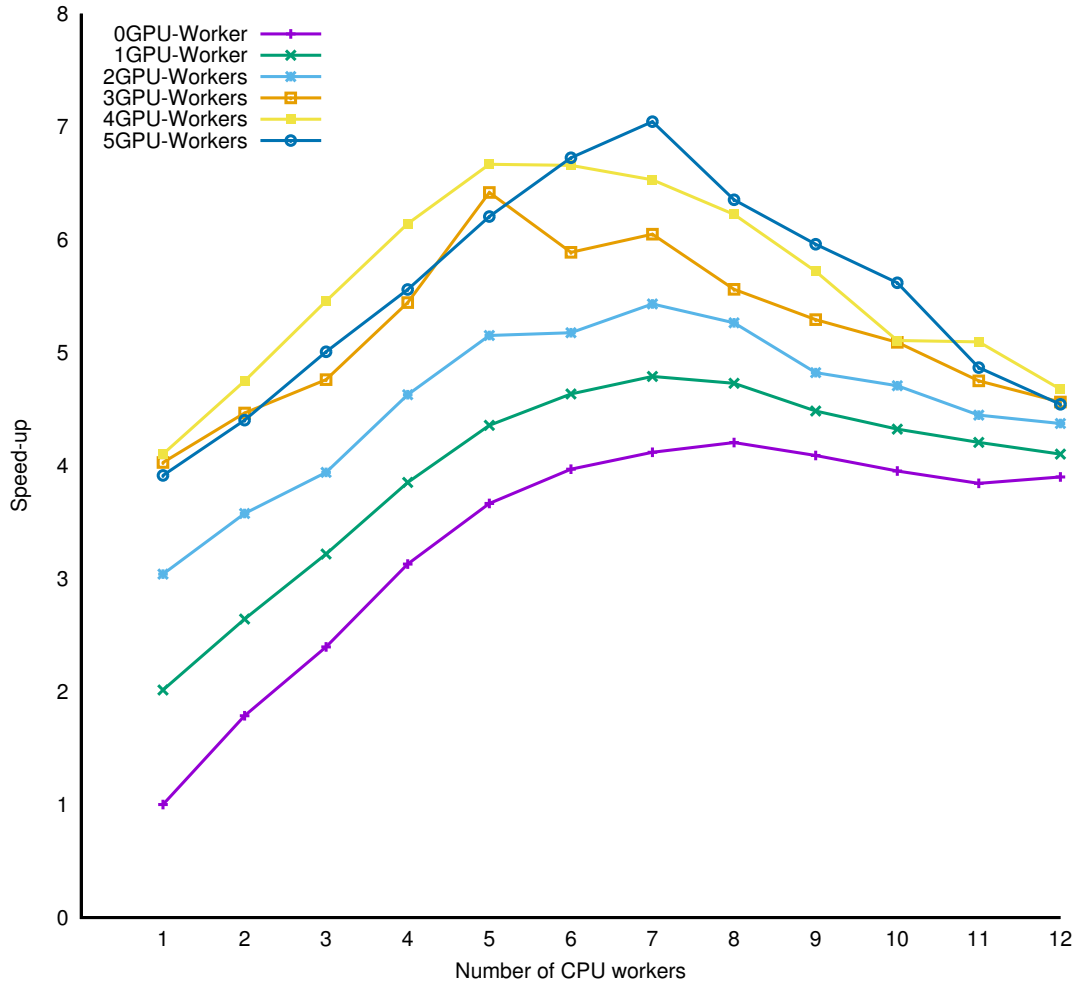


Figure 6.18: Speed-up graph for SMTWTP configurations.The evaluation of the SMTWTP application has been performed on the Titanic machine

Figure 6.17-a represents the visualisation of the abstract computation graph for SMTWTP and Figure 6.17-b demonstrates the MCTS decision tree generated for SMTWTP. The feedback process only affects the input rate for queue Q_1

Applying the static skeleton configuration for the farm pattern getSolution demonstrates that an optimum configuration is $\{findSol_{CPU} = 9; findSol_{GPU} = 5;\}$.

In order to further analyse the accuracy of the result, different possible configurations of the farm pattern are executed exhaustively. Figure 6.18 represents the farm speed-ups over the serial version for different configurations. Each line shows the speed-ups with a fixed number of GPU workers and varying the number of CPU workers. From the figure, we can observe that the best speed-up of 7.04 is obtained with $(7,5)$ CPU and GPU workers. The MCTS model predicted the best speed-ups for $(9,5)$ CPU and GPU workers, and for this mapping we obtained the speed-up of 5.95. Therefore, the static skeleton configuration gives the speed-up that is within 15% of the best obtained skeleton configuration. In the figure, we have omitted the

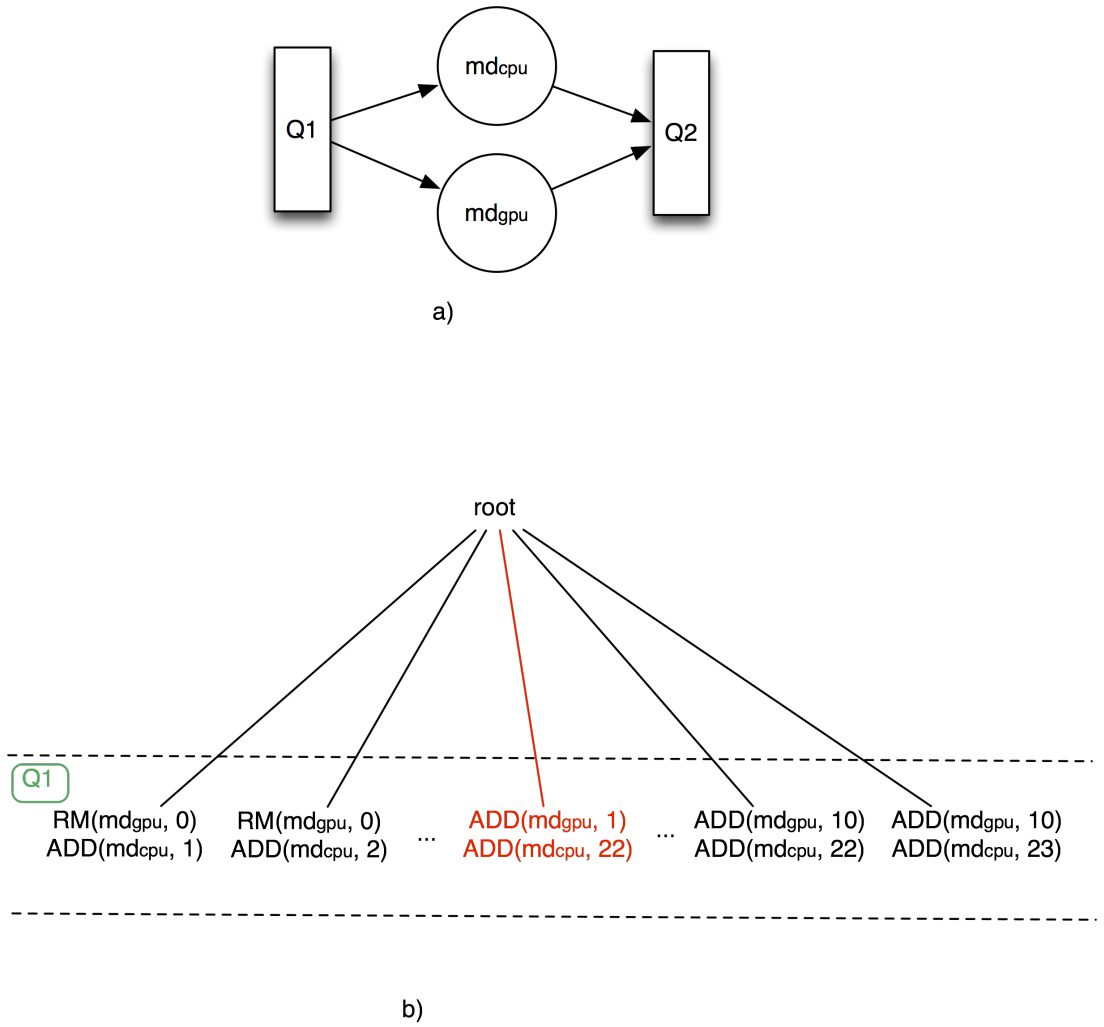


Figure 6.19: a) Abstract computation graph for MD. b) MCTS decision tree generated for graph a

speed-ups when more than 12 $getSolution_{CPU}$ workers are used (due to the NUMA architecture and the fact that our version of ACO is very data-intensive), as these speed-ups are smaller than when fewer CPU workers are used.

MD Using the reduction rules provided in equation 4.12, the MD abstract computation graph is:

$\wedge(\langle\langle md_{CPU} \rangle\rangle \vee \langle\langle\ md_{GPU}\ \rangle\rangle)\wedge.$

Figure 6.17-a represents the visualisation of the abstract computation graph for MD and Figure 6.19-b demonstrates the MCTS decision tree generated for MD.

The static skeleton configuration on the molecular dynamic farm pattern demonstrates that an optimum configuration is $\{md_{CPU} = 22; md_{GPU} = 1;\}$.

This means that the second farm can be changed to a $ParWrapper$ and therefore the appli-

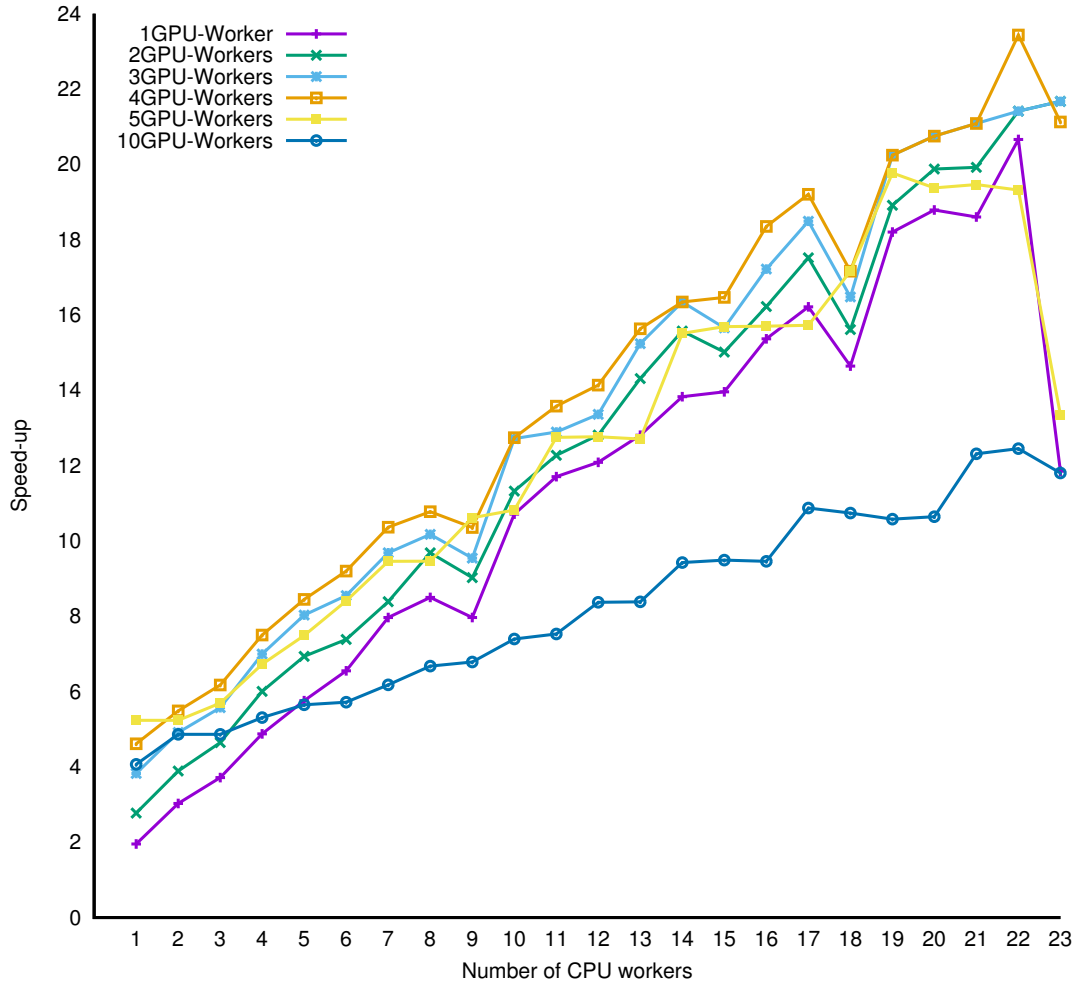


Figure 6.20: Speed-up graph for the molecular dynamics configurations.The evaluation of the MD application has been performed on the Titanic machine

cation can be re-written as:

$\square_{MD} ::= \Diamond_{MD}$

$\Diamond_{MD} ::= unicast_{1 \triangleleft 2} \,.\, [\bigcirc CPU_{sol}, \langle |\text{md\_gpu}| \rangle]$

$\bigcirc CPU_{sol} ::= scatter_{1 \triangleleft 22} \,.\, [\langle\langle \text{md\_cpu} \rangle\rangle]_{22} \,.\, gatherall_{22 \triangleright 1}$

To further analyse the accuracy of the result, the different possible configurations of the farm pattern are executed exhaustively. Figure 6.20 shows the speed-ups for a domain of 1000 molecules for different MD farm configurations. In the figure, the x axis corresponds to the number of CPU workers, and each line in the graph corresponds to a fixed number of GPU workers. In the figure, the best obtained speed-up for this configuration is 23.43 for 22 CPU workers and 4 GPU workers.

From Figure 6.20, we can see that the static skeleton configuration $(22, 1)$ gives us a speed-up of 20.65. The accuracy of the static skeleton configuration prediction for this application is therefore within 12% of the best possible speed-up obtained.

In both cases part of the deviation from the optimum results for static skeleton configurations is related to the slight differentiation in the estimated execution time by the MCTS simulation result and the actual execution time of the applications. However, in both cases, the acquired results are within the acceptable optimisation ranges.

6.4 Summary

In this chapter we have used PEI to improve the performance coordination of existing FastFlow applications.

In section 6.1, we have introduced 6 existing FastFlow applications that neither use *HWrapper*, nor *ParWrapper* components. Table 6.1 states the characteristics of these applications. These applications run only on multi-core CPU devices and do not use any GPU-devices. In each application, the FastFlow farm pattern is used to augment the level of parallelism on multi-core CPU devices.

In section 6.2, we have introduced 3 existing FastFlow applications that do not use *HWrapper* but use *ParWrapper* components. Table 6.2 states the characteristics of these applications. These three applications cover the heterogeneous pipeline composition (Pipeline pattern), heterogeneous reduction composition (Farm pattern with collector), heterogeneous non-reduction composition (Farm pattern without collector), feedback operation and nested patterns combinations supported by FastFlow. These application run on heterogeneous (CPU/GPU) multi-core architectures. However, component allocation for GPU-based components using *ParWrapper* must be handled manually as FastFlow sees the *ParWrapper* as a black box and has no information about the types of functions (OpenMP based, OpenCL-based or CUDA-based) embedded inside the *ParWrapper*.

In section 6.3, following evaluations have been performed to assess the adaptability and performance improvements of PEI based applications using *ParWrapper*. The evaluations have been carried out on three different heterogeneous multi-core architectures, namely, RGU Xookik cluster, the University of Pisa Titanic machine specified on Chapter 5, Table 5.3 and also Eddi located at Edinburgh Compute and Data Facility (specified at Table 6.3).

Performance Overhead: In section 6.3.1, we have considered two following types of overhead evaluation:

- *FastFlow Overhead:* In section 6.3.1.1, we have evaluated the overhead of the building block based approach on application scalability with highly specialised skeleton frameworks. FastFlow, SKePU and Thrust have been considered as candidate frameworks for this evaluation, where Fastflow represents a building block based framework and SKePU and Thrust represent the highly specialised skeleton

frameworks. The evaluation has been carried on Eddie machine. We have demonstrated that the Thrust version of *N*-body-Simulation for 1024 input data size (Figures 6.10) achieves linear speed-up (doubling the number of processors doubles the speed). However, the FastFlow version of *N*-body-Simulation for 8192 and 65536 applications (Figures 6.11 and 6.12) achieve linear speed-up and therefore has the least overhead when large-scale input data-size are applied. In this case, we have achieved up to 2 times more speed-up than Thrust and SkePU Framework for *N*-body simulation on large-scale input data size.

- *PEI Overhead:*In section 6.3.1.2, we have demonstrated that the minimum and maximum overhead of PEI framework for the proposed image processing applications are 1% and 3%, respectively (Figure 6.13). The overhead is related to the cost of extracting the sensor information dynamically with regards to the applied monitoring policy. Also, the minimum overhead can be used to determine a lower bound for improvements that should be achieved by the coordination optimisation methods.

Efficient Idling: In section 6.3.2 the proposed efficient Idling technique, presented in chapter 4, section 4.1.4 has been used to improve the performance of FastFlow benchmark applications by efficiently utilising the underlying CPU device cores. Not only We have achieved up to 27% speed-up by using efficient idling technique, but also we have prevented up to 200% drop in application speed-up when the number of components are more than the number of CPU cores (Figure 6.14).

Memory Management: In section 6.3.3 we have applied our dynamic memory management technique to adaptively control the memory usage for the custom implementation of Eispack routines as a large-scale heavy workload application (Figure 6.16). The application benefits from the dynamic memory management technique that is implemented in PEI to prevent queue overflows for pipeline components. This section demonstrates that our proposed memory management technique is crucial to allow FastFlow to scale up to large problems, as earlier attempts at using the FastFlow Pipeline pattern **without** using memory management technique resulted in a fatal system error due to insufficient memory space.

Static Structural Configuration: In section 6.3.4, we have applied the static structural configuration proposed in PEI to improve the performance of MD and SMTWTP applications on Titanic machine. Our evaluation demonstrate that we have achieved up to 20 times speed-up when using the proposed static structural configuration to adjust the number of workers for molecular dynamic application (figure 6.19) on Titanic machine. We have also achieved up to 5 times speed-up when using the proposed static structural

configuration to adjust the number of worker for SMTWTP application (figure 6.17) on Titanic machine.

Chapter 7

Conclusion & Future Work

In this chapter we have reviewed our research objectives, followed by our research impact in the research landscape. Possible future directions have also been highlighted as a continuation of this research.

7.1 Consolidation of Research

In this thesis we have empirically demonstrated the validation of our hypothesis by developing an autonomic behavioural framework, called PEI, that manifests our SKIP methodology synthesised with RISC-pb^{2}l libraries.

PEI is composed of 3 following main parts:

Structured Parallel Programming Model that separates coordination from computation and hide the synchronisation and communications between application's components from end-users.

We have selected the existing RISC-Pb^{2}l building block approach as a structured parallel programming model. In Chapter 3, section 3.2, We have extended the RISC-Pb^{2}l by adding a new heterogeneous block that enables the coordination of application over heterogeneous multi-core architecture.

In Chapter 3, section 3.2, we have also designed a grammar for the extended RISC-Pb^{2}l that detects the set of parallel programming patterns supportable by RISC-Pb^{2}l building block library for heterogeneous multi-core architecture. The grammar checks the correction and validation of application's structure generated by RISC-Pb^{2}l library.

We have employed FastFlow as an existing RISC-Pb^{2}l framework for homogeneous multi-core system. In Chapter 4, section 4.1.1, we have extended the FastFlow by implementing a GPU back-end that supports application execution over heterogeneous multi-core architecture. This expansion is the implementation of the new heterogeneous

building block. We call it HFastFlow.

High-level Abstraction Layer (HAL) to support autonomic management for structured application over heterogeneous multi-core architecture. Presented in Chapter 4, section 4.3, HAL includes three interfaces:

1. *User-level interface*: Determines a set of extra-functional and non-functional properties that allows end-user to descriptively define the structural composition of its application. It also allows the end-user to descriptively determine the constrain configuration of the application and the underlying architecture.
2. *System level interface*: Determines a set of extra-functional and non-functional properties that are *i)* monitored on the structured parallel framework; *ii)* monitored on underlying resources; *iii)* addressed by coordination engines.
3. *Autonomic manager interface*: Is a bridge between end-user, the structured framework, coordination engines, and underlying hardware. It can automatically apply the specific coordination decisions regarding to adapt to the changes happens on both application status or underlying hardware status.

Followings are the steps to design and develop HAL.

In Chapter 3, section 3.1, we have determined a set of controlling parameters that can affect application coordination. controlling parameters have been extracted from investigating Thrust, SKePU, FastFlow and Intel TBB frameworks that support coordination over heterogeneous architectures.

In Chapter 3, section 3.3, we have designed SKIP as a generic methodology that enables autonomic management for structured parallel programming model over heterogeneous multi-core architectures. It determines both user-level and system level interfaces required to provide an autonomic management over a structured parallel application. We have used SKIP with the RISC-Pb^{2}l grammar to generate the high-level abstraction layer for RISC-Pb^{2}l building block approach.

In Chapter 4, section 4.2, we have instrumented HFastFlow by adding a set of actuators and sensors to exchange the extra-functional and non-functional properties determined in SKIP. This instrumentation represents the implementation of the SKIP fusion on RISC-Pb^{2}l.

In Chapter 4, section 4.3.1, we have designed and implemented a SKIP adaptor that autogenerate a RISC-Pb2l application from a descriptive structural composition file provided by end-user.

In Chapter 4, section 4.3.2, we have designed and implemented a dynamic runtime interface (DSRI) to provide the autonomic management on HFastFlow for heterogeneous

multi-core architecture.

Performance Enhancement Tools (PETs) is a set of coordination engines that supports scheduling, load balancing and static configuration of an application structure for a specific heterogeneous multi-core architecture. We have designed and developed these coordination engines as they have been considered as key optimisation objectives for ParaPhrase project [40].

We have designed and implemented a set of coordination engines in order to verify the suitability of HAL for orchestrating structured parallel applications over heterogeneous architecture. These engines interact with HFastFlow through the autonomic manager (DSRI) to optimise applications performance and resource utilisation over heterogeneous architecture. Followings are the coordination engines we have designed and implemented in this thesis.

In Chapter 4, section 4.4.3 we have designed and developed a scheduling mechanism to allocate/reallocate different components of applications on underlying heterogeneous devices. It can dynamically remap heterogeneous software components to the available CPU/GPU devices based on information provided by the high-level virtualisation interfaces about the extra-functional properties of the software components, on the hardware performance characteristics, and on information that is obtained by monitoring the dynamic system load. The provided system is capable of dealing with components from multiple applications and remapping them to the best available hardware, so ensuring optimal use of the available hardware resources.

In Chapter 4, section 4.4.2 we have designed and developed a load-balancing technique that auto-tune the workload fraction over different heterogeneous components with regards to their computational power. It will also respond to the sudden changes on the input workload size and if required, automatically tune the workload fraction for application's components.

In Chapter 4, section 4.4.4 we have designed and developed a static structural configuration technique to tune an application structure for a specific heterogeneous architecture. As an external coordination method, the static structural configuration tries to automatically tune a structured application and to map its components to the available resources for a given architecture in order to maximise the application throughput.

In Chapter 4, section 4.1.4 we have designed and developed an efficient idling technique to maximum efficient utilisation of the resources for any CPU slot allocated to a component. The efficient utilisation of a resource depends on the availability of a task in a component queue for the allocated slot for the component.

In Chapter 4, section 4.1.3 we have designed and developed a memory management technique to dynamically control the memory usage for heavy workload application in order to prevent queues overflows for application components in HFastFlow.

The applicability and efficiency of our methodology has been verified through 15 applications divided into following categories:

Heterogeneous OpenCL Applications: The first category includes 6 new heterogeneous applications (presented in Chapter 5, section 5.1) that use the proposed *HWrapper* (OpenCL back-end in FastFlow) to develop components that run on GPU devices. These applications cover different RISC-Pb^{2}l building block compositions supported for FastFlow.

Homogeneous Applications: The second category includes 6 different existing homogeneous applications (presented in Chapter 6, section 6.1) that run on multi-core CPU devices and do not use GPU devices. These applications neither have *HWrapper*, nor *ParWrapper*.

Heterogeneous Generic Applications: The third category includes 3 different existing heterogeneous applications (presented in Chapter 6, section 6.2) that use *ParWrapper* for developing components that run on GPU devices.

Using PEI, we have achieved the following objectives.

GPU Utilisation: Using the proposed OpenCL back-end for developing GPU components for FastFLow applications, it is possible to increase the GPU utilisation by 38% (from 57% to 95%) for Heterogeneous OpenCL applications. We have also demonstrated that by using the proposed OpenCL back-end we can achieve up to 1 order of magnitude speed-up over serial CPU version of applications and Up to 1.68 times speed-up over serial GPU version of applications. The results are presented in Chapter 5, section 5.2.2.

Optimising OpenCL Component Scheduling: We have demonstrated that by using the proposed OpenCL Scheduling policy, it is possible to dynamically allocate/re-allocate the OpenCL component to OpenCL-enable devices in order to increase the performance. We have demonstrated that by using the proposed OpenCL scheduler we can achieve up to 1 order of magnitude speed-up over FastFlow round-robin scheduler. This result is presented in chapter 5, section 5.2.4.

Optimising Workload Distribution: By using the proposed adaptive load-balancer which is periodically update by the adaptive workload distribution tool, we can achieve up to 9 times speed-up over FastFlow round-robin load-balancer. The results are presented in Chapter 5, section 5.2.3.

Multi-Tenant Application Execution: The concurrent executions of Heterogeneous OpenCL applications have demonstrated that the overhead of switching resources between components is negligible. Moreover, we have achieved up to 2% improvement on total execution time of applications. Although executing multi-tenant applications in a concurrent mode may not significantly improve the performance, it provides extra flexibility for the application executions. Therefore, by prioritising the applications, small-scale applications can temporarily borrow resources from the large-scale ones without terminating the large-scale application. The results are presented in Chapter 5, section 5.2.5.

Low Performance Overhead: We have demonstrated that PEI overhead is around 3% for both new and existing applications when the aggressive sensor monitoring is applied. Moreover, when the sparse sensor monitoring (which is the default sensor monitoring policy) is applied, the PEI overhead is around 1%. The results are presented in Chapter 5, section 5.2.1 and Chapter 6, section 6.3.1.2. In comparison with the up to one order of magnitude performance improvement that we have achieved, we can say that the PEI overhead can be negligible.

Efficient Idling Optimisation: We have demonstrated that although the Homogeneous applications do not have any GPU components, they can still benefit from some of the PEI coordination engines such as efficient idling technique presented in Chapter 4, section 4.1.4. By using the efficient idling technique, We have demonstrated that it is possible to achieve up to 2 times speed up for these applications when the number components are more than the number of available CPU cores on the underlying architecture. The results are presented in Chapter 6, section 6.3.2.

Controlling Memory Management Dynamically: We have demonstrated that the proposed dynamic memory management technique can adaptively control the memory usage for the custom implementation of EISPACK routines as a large-scale heavy workload application. The application benefits from the dynamic memory management technique that is implemented in PEI to prevent queue overflows for pipeline components. We have demonstrated that using the memory management technique is crucial to allow FastFlow to scale-up for large problems, as earlier attempts at using the FastFlow Pipeline pattern **without** using memory management technique resulted in a fatal system error due to insufficient memory space. The results are presented in Chapter 6, section 6.3.3

Statically Optimising Application Structural Configuration: We have applied the static structural configuration technique proposed in Chapter 4, section 4.4.4 to improve the performance of both existing heterogeneous applications and heterogeneous OpenCL applications. We have demonstrated that we can achieve up to 5 times speed-up for recursive Gaussian application when using the static structural configuration technique over

FastFlow round-robin technique for mapping components. This result is presented in Chapter 5, section 5.2.4. We have achieved up to 20 times speed-up when using the proposed static structural configuration to adjust the number of workers for molecular dynamic application on Titanic machine and map the workers for the CPU/GPU devices in this machine. We have also achieved up to 5 times speed-up when using the proposed static structural configuration to adjust the number of worker for SMTWTP application on Titanic machine and map the workers for the CPU/GPU devices in this machine. The results are presented in Chapter 6, section 6.3.4

7.2 Research Impact

This thesis was part of the Paraphrase project,. The project has produced a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component based applications that can be mapped to the available hardware resources and, if required, dynamically re-mapped to meet the application needs and hardware availability. As part of ParaPhrase project, this thesis has successfully delivered the following Paraphrase objectives:

GPU Support for FastFlow: Presented in Chapter 4, section 4.1.1, we have extended FastFlow by implementing an OpenCL back-end that supports application execution over heterogeneous multi-core architectures. We call this expansion HFastFlow which is the implementation of the *HWrapper* building block presented in Chapter 3, section 3.2. Using the proposed OpenCL back-end for developing GPU components in HFastFLow, it is possible to increase the GPU utilisation by 38% (from 57% to 95%) for Heterogeneous OpenCL applications. We have also demonstrated that by using the proposed OpenCL back-end we can achieve up to one order of magnitude speed-up over serial CPU version of applications and up to 1.68 times speed-up over the serial GPU version of applications. The results are presented in Chapter 5, section 5.2.2.

High-Level Virtualisation Layer: We have designed and developed ODVL, presented in Chapter 4, section 4.3.3, as a virtual representation of all available OpenCL capable devices for the underlying heterogeneous multi-core systems. ODVL represents a unified view of all available OpenCL-enabled devices independent from device vendors. The proposed OpenCL scheduler presented in Chapter 4, section 4.4.3, uses ODVL for dynamic mapping/remapping of components to the OpenCL capable devices and switching components between these devices.

Static Mapping: Through static structural configuration presented in Chapter 4, section 4.4.4, we have designed and developed a static mapping for OpenCL components and the avail-

able hardware resources collectively represented by the ODVL. The static structural configuration heuristically maps application components based on historical and static extra-functional information that is extracted from the instrumented framework in a SKIP format for the coordination component interface with regards to the performance characteristics that are exposed to the applications and heterogeneous-multi-core architectures. The result presented in Chapter 5, section 5.2.4 demonstrates that we have achieved up to 20 times speed-up when using the proposed approach to generate static mapping for molecular dynamic application on Titanic machine. Also, the results are presented in Chapter 6, section 6.3.4 demonstrate that we achieved up to 5 times speed-up when using the proposed approach to generate static mapping for the SMTWTP application on Titanic machine.

Dynamic Mapping/Re-Mapping: Using the proposed OpenCL Scheduler presented in Chapter 4, section 4.4.3, it is possible to dynamically remap the application components to the available resources. The OpenCL scheduler uses the extracted sensor file containing the application performance metrics and the information obtained by monitoring the dynamic system load. In Chapter 5, section 5.2.4, we have demonstrated that by using the proposed OpenCL scheduler on heterogeneous OpenCL applications, we can achieve up to one order of magnitude speed-up over HFastFlow round robin scheduler.

Priority-based Multi-Tenant Applications Executions: Presented in Chapter 4, section 4.4.3.1, we have provided a priority based system type that is capable of classifying and controlling the priority of applications. As part of the proposed OpenCL scheduler, the priority system will allow some basic quality of service capabilities, where small-scale applications with high priority can get better throughput by stealing the resources from large-scale, low priority applications without terminating them. Therefore, by using the priority system, the OpenCL scheduler is capable of executing multi-tenant applications where different components from multiple applications map/remap to the best available hardware, based on the application priority. In Chapter 5, section 5.2.5, we have demonstrated that by prioritising the heterogeneous OpenCL applications, small-scale applications can temporarily borrow resources from the large-scale ones with close to 0% switching overhead and without terminating the large-scale application.

Besides ParaPhrase partners applications: EISPACK Routine, MD and SMPWTP applications (presented in Chapter 6, section 6.2) that use PEI for performance enhancement, other ParaPhrase partners have also used this research. We will explain some of them in the following:

In [115] an extension of HFastFlow over the distributed architecture has been introduced to support a network of multi-core workstations. The proposed extension provides a structural

coordination of a programme in a single workstation while the distribution of the data across the stations has been delegated to the ZeroMq messaging system, which is encapsulated by a class called `ff_dnode`. Sub-classing `ff_node`, `ff_dnode` is responsible for connecting the edge component of the application running in one station, with one or more edge components of the application running on the same or different workstations. However, the data marshalling and unmarshalling processes across the workstations are left unattended.

In [116] a systematic methodology has been proposed to exploit the approximated analytical cost models of applications in HFastFlow. The proposed approach has been incorporated with an integrated programming framework to target both local and remote resources to support the offloading of computations from structured parallel applications to heterogeneous cloud resources. Applying such a system, it is possible to map a data intensive application on a combination of local and remote resources to guarantee optimal performances. Also, it is possible to calculate the optimal proportion of cloud resources to achieve a given target performance value.

In [117] a deployment tool has been provided for the on-demand distribution of different instances of HFastFlow applications on the cloud computing system using the Open Virtualisation Format. Using SKIP compliant extracted structural information and performance metrics, the provided tool automatically determines an optimum hardware/system software configuration of a new instance of an instrumented HFastFlow framework among the available heterogeneous cloud resources. This tool registers itself as a PET for PEI in order to use the extracted sensor information.

In [118] a simple analytical performance model is developed for HFastFlow that supports the autonomic offloading of sup-tasks in data parallelism skeleton patterns performing on hybrid CPUs and GPUs. By applying the provided model, sub-tasks are divided into two partitions, one to be computed on the GPU and the other on the multi-core CPU. When the percentage of sub-tasks determined for a multi-core CPU is less than a predefined threshold, all subtasks are offloaded to the GPU. Although the model is not entirely accurate, it reasonably predicts the potential improvement of an application runtime when using the hybrid execution of CPU and GPU cores.

In [82] a new refactoring tool has been designed to increase the programmability of parallel systems. The presented refactoring tool provides a set of formal pattern rewriting rules to translate the potential parallelisable blocks of a serial application into Farm and Pipeline patterns in HFastFlow. The provided tool requires the assistance of a programmer to detect and verify the parallelisable blocks. The refactoring tool has been integrated with the static structural configuration to further tune the application structure and static mapping for a specific heterogeneous architecture.

In [119], an implementation of the proposed *HWrapper* for CUDA library has been implemented for FastFLow. The CUDA back-end allows developer to embed a block of CUDA code as a component of supported parallel patterns in FastFlow. The CUDA back-end supports the execution of heterogeneous applications over heterogeneous (CPU/GPU) multi-core architecture.

7.3 Ongoing Research and Future Work

The SKIP compliant autonomic framework presented here covers the coordination aspect of parallel programming over heterogeneous multi-core systems.

As stated in Chapter 4, currently HFastFlow only supports the skeleton-based parallel patterns through RISC-pb^{2}l. Therefore, although the SKIP methodology can demonstrate and construct such patterns (it has been represented in Appendix A), full operability of all the compositional features of RISC-pb^{2}l libraries will not be available. In this case, as an ongoing research we are investigating the applicability of extending this work to implement the remaining RISC-pb^{2}l supported patterns for HFastFlow to enable all the compositional features of RISC-pb^{2}l and consequently to expand the provided SKIP adaptor to construct and instrument them.

One possible future extension could address the dynamic refactoring of an application through SKIP. Considering an application as a composition of building blocks executing on a set of resources, it is possible to merge or split the components using the optimisation rules provided in [15]. In this case, any combination or separation of building blocks can change the structure of an application and resource utilisations. Therefore, an optimisation tool is required to detect and substitute the equivalent pattern. Incorporating with the SKIP adaptor, such an optimisation tool could automatically refactor an application in order to improve the performance. Knowing the composition of a component can be reversible, by periodically assessing an application's performance through sensor information, it would be possible to preserve the original structure in case of a drop in performance. Therefore, an application can automatically adapt to different environmental variations. Such an expansion or shrinking of an application can be useful for multi-tenant application executions especially when there is a high demand for resource access.

Another direction to expand this research could be to embed different instrumented frameworks in the system. Applying SKIP and skeleton adaptor concepts, we provide a clean separation between the parallel coordination patterns and computation functions executed by these patterns. Using this technique provides the possibility of having framework independent applications. Using a SKIP compliant descriptive object as a front-end language, different frameworks that support the building block approach can be considered as back-end parallel struc-

tural frameworks where each back-end provides its own translation of the SKIP compliant descriptive object by implementing its skeleton adaptor. Having multiple back-ends, a heuristic algorithm can be provided to choose the most appropriate back-end framework for different criteria. Such criteria can include environmental constraints, resource availability, application structure, application scale, and resource usage.

Another possible extension would be to provide a fault-tolerant PEI which extends the extra-functional properties covered in this thesis. To address the client-crash in the system, one possible action could be the "respond time-out" mechanism. In this case a server can periodically request an acknowledgement from its registered clients. If no response is received from a client in the determined time-out period, the server can terminate the client's application and release its resources. To address the server crash issue, one possible solution could be to provide a mirror server. A mirror server will backup the main DSRI server and it continuously synchronises itself with the main server. In the case of a crash, the main sever can be replaced by the mirror one.

Another potential future direction for this work is to expand the PEI for distributed systems, either in the form of a cluster of heterogeneous multi-core systems or in the form of distributed workstations on the cloud. When the components of an application are scattered throughout the workstations, having a fault-tolerant PEI is crucial. In this case, in addition to the fault-tolerant support mentioned above, a mechanism must exist to notify the root workstation (master-workstation) of the system about the crash in one or more of the stations. One possible action in this case could be to re-launch the crashed component in a new workstation. Also, the mirrored technique mentioned above can be used to address the crash in the root workstation.

One possible direction to expand the PEI for distributed systems can be to provide a hierarchical DSRI architecture by adding an extra coordinator layer. In this case the workstation containing the coordinator can be considered as the master workstation that manages all the DSRI servers on different workstations as its slaves. Figure 7.1 represents the architectural view of the distributed PEI.

Finally, data transferring latency on the network should be considered in coordination engines. Such a factor must be taken into account in both the workload distribution and allocation of components to different workstations in a distributed system. The distributed PEI also requires an on-demand launcher for the allocation of components to different workstations. In this case, while each server has some level of autonomy to control the registered applications in its own workstation, the DSRI coordinator launches/re-launches components in a new workstation for an application. Also, the DSRI coordinator would be responsible for re-launching a DSRI server in each workstation in case of a DSRI server crash.

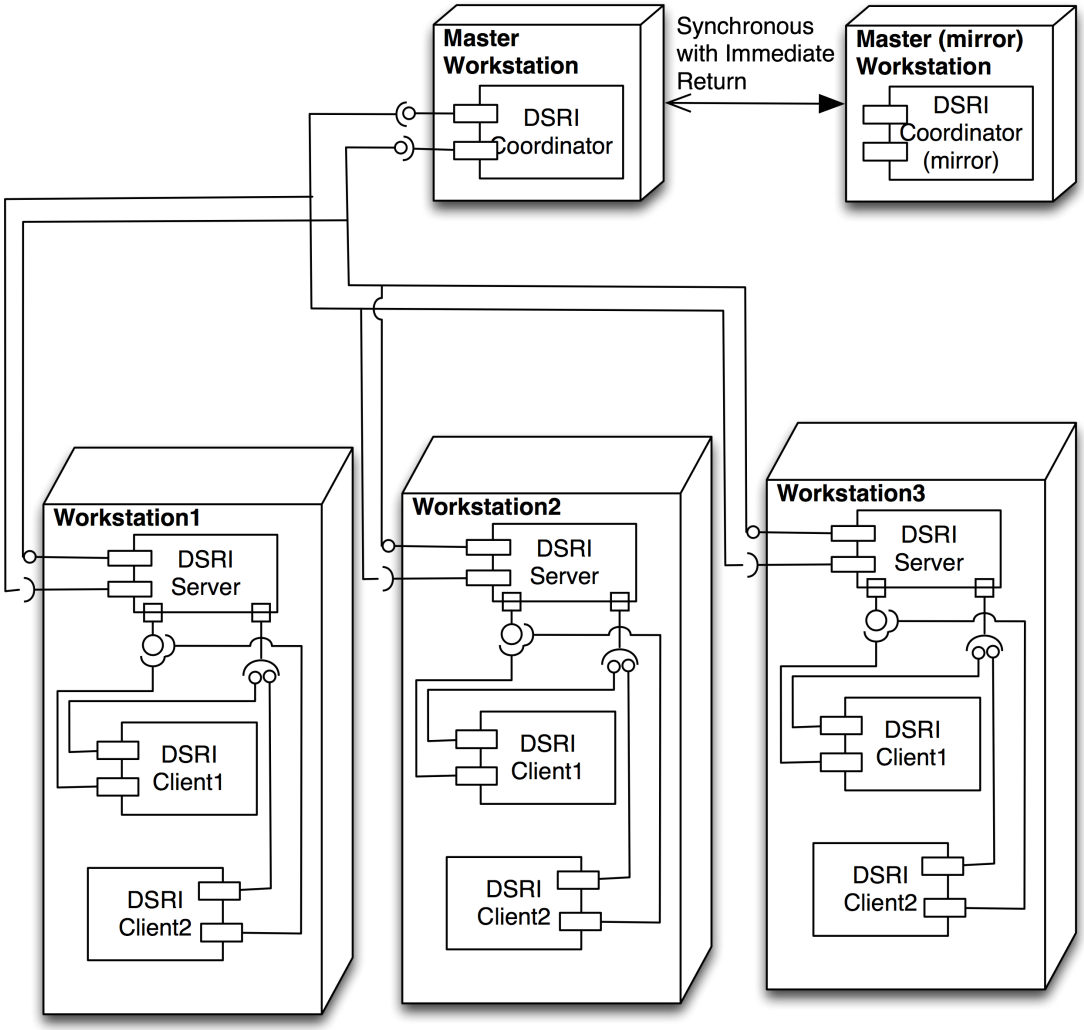


Figure 7.1: Deployment diagram for distributed PEI

.

Appendix A

Validation of RISC-pb^{2}l Grammar

In the following we explain how the RISC-pb^{2}l grammar represented in Listing 3.1 captures the supported patterns presented in [14, 15].

A.1 Skeleton-based Parallel Patterns

Here, we demonstrate that the skeleton based patterns supported by RISC-pb^{2}l are detectable by RISC-pb^{2}l grammar.

A.1.1 Embarrassingly Parallel Patterns

Embarrassing parallel patterns include *Task-Farm*, *Map*, and *MISD*. Although the syntax for these three patterns are the same, the difference among them is around the applied policy to the combinator building blocks for spreading or gathering the data.

In *Task-Farm* pattern usually the *unicast* policy is used as a data distribution policy for 1*toN* combinator and *gather* policy is used as a date collection policy for *Nto*1 combinator. In *Map* pattern usually the *scatter* policy is used as a data distribution policy for 1*toN* combinator and *gatherall* policy is used as a data collection policy for *Nto*1 combinator. In *MISD* pattern the *broadcast* policy is used as a data distribution policy for 1*toN* combinator.

$\square_{ep} ::= \blacktriangle_{ep}$

$\blacktriangle_{ep} ::= \triangle_{ep}$

$\triangle_{ep} ::= \dashv^{1n} . \pentagon^{n} . \vdash^{n1}$

$\pentagon^{n}$ can represent any nested computation required for the embarrassing parallel pattern.

It is also possible to use the non-reduction composition that eliminates the *Nto*1 combinator. In this case the patterns can be captured by the following.

$\square_{ep} ::= \blacklozenge_{ep}$

$\blacklozenge_{ep} ::= \lozenge_{ep}$

$\lozenge_{ep} ::= \dashv^{1n} . \bigcirc^{n}$

$\bigcirc_n ::= \pentagon_n$

Moreover, the following composition is detectable by the grammar provided in Listing 3.1 to support feedback for embarrassing parallel pattern, where $\pentagon$ can be replaced by any valid compositions.

$\square_{ep} ::= \blacktriangle_{ep}$

$\blacktriangle_{ep} ::= \triangle_{ep}$

$\triangle_{ep} ::= \overleftarrow{\left(\dashv^{1n} . \pentagon_{ep}{}^{n} . \vdash^{n1}\right)_c}$

When the non-reduction composition is applied the feedback building block can be detected as follows.

$\square_{ep} ::= \blacklozenge_{ep}$

$\blacklozenge_{ep} ::= \overleftarrow{\diamond_{ep}}$

$\diamond_{ep} ::= \overleftarrow{\left(\models_1^{n1} . \dashv^{1n} . \pentagon^{n}\right)_c}$

In the feedback based embarrassing parallel pattern proposed above, it is possible to implement *while* and *for* patterns by replacing the combinator's distribution policy function with a custom function for checking the condition.

Furthermore, when *MISD* computation is applied the *if* pattern can be detected by implementing a custom function f in $\left(f_{1 \lhd n}\right)$.

A.1.2 Reduction

A reduction combines every element in a collection into a single element using an associative combiner function [49]

The reduction pattern can be detected via the following instructions by the provided grammar:

$\square_{\oplus} ::= \blacktriangle_{\oplus}$

$\blacktriangle_{\oplus} ::= \triangle_{\oplus}$

$\triangle_{\oplus} ::= \dashv^{1n} . \vdash^{n1}$

$\dashv^{1n} ::= scatter_{1 \lhd n}$

$\vdash^{n1} ::= \vdash_{mk}^{n1}$

$\vdash_{mk}^{n1} ::= \left(\vdash^{m1}\right)^{k} . \vdash^{k1}$

where $\vdash^{k1}$ can be determined recursively as $\left(\vdash^{x1}\right)^{y} . \vdash^{y1}$ until the required level of parallelism for reduction pattern in satisfied.

Moreover, in order to add a custom computation function during reduction process the last instruction above can be replaced by $\vdash_{mk}^{n1} ::= \left(\vdash^{m1}\right)^{k} . \pentagon^{k} . \vdash^{k1}$.

A.1.3 Pipe

The *pipe* pattern can be generated by the following instructions.

$\square_{Pipe} ::= \blacktriangle_{Pipe}$

$\blacktriangle_{Pipe} \ ::= \ \triangledown_{Pipe}$

$\triangledown_{Pipe} \ ::= \ \triangle_{Pipe} \centerdot \triangledown_{Pipe} \mid \triangle_{Pipe} \centerdot \triangle_{Pipe}$

Replacing the right $\triangledown_{Pipe}$ in the third line with $\triangle_{Pipe} \centerdot \triangledown_{Pipe}$ can generate arbitrary number of stages in *pipe* pattern. It is important to note that the minimum number of stage in *pipe* is $2(\triangle_{Pipe} \centerdot \triangle_{Pipe})$. Replacing $\triangle$ for each stage with any valid represented composition will support a nested patterns.

The last stage in a *pipe* pattern has more flexibility as there is no need to feed data into the *pipe*. Therefore, we can generate the *pipe* pattern where its last stage consists of a nun-reduction composition, as represented in the following.

$\square_{NoRPipe} \ ::= \ \blacklozenge_{NoRPipe}$

$\blacklozenge_{NoRPipe} \ ::= \ \blacktriangle_{NoRPipe} \centerdot \blacklozenge_{NoRPipe}$

The following combination of a *pipe* pattern with feedback is also detectable by the proposed grammar, where the last stage of the *pipe* pattern is limited to a reduction composition.

$\square_{Pipe} \ ::= \ \blacktriangle_{Pipe}$

$\blacktriangle_{Pipe} \ ::= \ \triangle_{Pipe}$

$\triangle_{Pipe} \ ::= \ \overleftarrow{(\triangledown_{Pipe})_c}$

$\triangledown_{Pipe} \ ::= \ \triangle_{Pipe} \centerdot \triangledown_{Pipe} \mid \triangle_{Pipe} \centerdot \triangle_{Pipe}$

In order to have a feedback *pipe* pattern with a non-reduction composition, the following instruction can be used.

$\square \ ::= \ \diamondsuit$

$\diamondsuit \ ::= \ \overleftarrow{(\models_1^{n1} \centerdot \blacktriangle \centerdot \dashv^{1n} \centerdot \pentagon^n)_c}$

Moreover, it is possible to have a feedback for a set of stages of a *pipe* pattern by applying $\triangle_{substages} \ ::= \ \overleftarrow{(\triangledown_{substages})_c}$ for any specific $\triangle$ and replacing the $\triangledown_{substages}$ with $\triangledown_{substages} \ ::= \ \triangle_{substages} \centerdot \triangledown_{substages} \mid \triangle_{substages} \centerdot \triangle_{substages}$ for the set of determined stages. There is no self-feedback for a stage in a *pipe* pattern, as a self feedback stage can be implemented by a recursive function, which would be faster than using a feedback based queueing system for a self-feedback stage.

A.1.4 Divide & Conquer

The divide & conquer pattern is composed of two steps, namely, divide and conquer. The first step divides the task into sub-tasks until the base case condition is satisfied. In the next step the generated sub-tasks are partially conquered according to the unique indexing number allocated to them. This will continue until the results are reduced to a single one. The spread function (f_{con}) directs all tasks with the same indexing number into a single computing unit for the reduction stage. The following instructions demonstrate the divide & conquer pattern captured by the proposed grammar.

$\square_{div\&con} \ ::= \ \blacktriangle_{div\&con}$

$\blacktriangle_{div\&con} \ ::= \ \triangledown_{div\&con}$

$$\nabla_{div\&con} ::= \triangle_{div} \cdot \triangle_{con}$$

$$\triangle_{div} ::= \overleftarrow{\left(\dashv^{1n} \cdot \pentagon_{div}{}^{n} \cdot \vdash^{n1}\right)_{c_1}}$$

$$\pentagon_{div}{}^{n} ::= \left(\blacktriangle_{divcal}\right)^{n}$$

$$\blacktriangle_{divcal} ::= \triangle_{divcal}$$

$$\triangle_{divcal} ::= \dashv^{1p} \cdot \vdash^{p1}$$

$$\dashv_{1p} ::= \left(f_{div\,1 \lhd p}\right)$$

$$\vdash_{p1} ::= gather_{p \rhd 1}$$

$$\dashv_{1n} ::= unicast_{1 \lhd n}$$

$$\vdash_{n1} ::= gather_{n \rhd 1}$$

$$\triangle_{con} ::= \overleftarrow{\left(\dashv^{1x} \cdot \vdash^{x1}\right)_{c_2}}$$

$$\dashv^{1x} ::= \dashv^{1x}_{mk}$$

$$\dashv^{1x}_{mk} ::= \left(\dashv^{1m}\right)^{k}$$

$$\vdash^{x1} ::= \vdash^{x1}_{mk}$$

$$\vdash^{x1}_{mk} ::= \left(\vdash^{m1}\right)^{k} \cdot \vdash^{k1}$$

$$\dashv^{1m} ::= \left(f_{con\,1 \lhd m}\right)$$

$$\vdash^{m1} ::= \left(reduce_{con\,m \rhd 1}\right)$$

$$\vdash^{k1} ::= gather_{con\,k \rhd 1}$$

A.1.5 Stencil

In *stencil* programming pattern, data structure B is derived by processing data structure A where each element of B is the result of processing a function f on the neighbourhood of the corresponding element in A. The stencil pattern is often iterated up to a certain point until the condition c_1 is verified.

Therefore two steps can be considered for calculating the neighbourhood for each element B and applying the function f for each element on the specified area.

The composition provided for *stencil* pattern in [14] is detectable by the proposed grammar as represented in the following.

$$\square_{stencil} ::= \blacktriangle_{stencil}$$

$$\blacktriangle_{stencil} ::= \triangle_{stencil}$$

$$\triangle_{stencil} ::= \overleftarrow{\left(\dashv_{broadcast_{1 \lhd n}} \cdot \pentagon^{n}{}_{stencil} \cdot \vdash_{gatherall_{n \rhd 1}}\right)_{c_1}}$$

$$\pentagon^{n}{}_{stencil} ::= \pentagon^{n}{}_{nbh} \cdot \pentagon^{n}{}_{f}$$

$$\pentagon^{n}{}_{nbh} ::= \left(\blacktriangle_{nbh}\right)^{n}$$

$$\blacktriangle_{nbh} ::= \triangle_{nbh}$$

$$\left(\triangle_{nbh}\right)^{n} ::= \left[\triangle_{nbh}\right]_{n}$$

$$\pentagon^{n}{}_{f} ::= \left(\blacktriangle_{f}\right)^{n}$$

$$\blacktriangle_{f} ::= \triangle_{f}$$

$$\left(\triangle_{f}\right)^{n} ::= \left[\triangle_{f}\right]_{n}$$

A.2 General Purpose Computing Models

In this section we demonstrate that the *BSP*, *Map-Reduce* and *MDF* patterns supported by RISC-pb^{2}l building blocks are also detectable by the proposed grammar.

A.2.1 *BSP*

Bulk Synchronous Parallel Model (BSP) [120] is a parallel computation that runs on a set of processors in a sequence of super-steps.

The BSP algorithm consists of the following steps:

- Each processor computes its allocated tasks locally by only accessing its local variables and environment.
- Eventually, processors initiate an asynchronous communication step to exchange data.
- Each processor enters a barrier waiting for the completion of the communication step.

In [15] the *BSP* pattern is divided into p supper-steps where each super-step is itself divided into n step. Each step is composed of a two-stage MISD building block to process the allocated task and reroute the result to the next supper-step. Using the RISC-pb^{2}l building blocks, the *BSP* pattern is represented as: $BSP = broadcast_{1\lhd n} \centerdot {ss_p}^n ... \centerdot {ss_i}^n ... \centerdot {ss_p}^n \centerdot gatherall_{n\lhd 1}$ where a super-step ss^i can be generated as follows.

$ss_i = [\triangle_{stp_{i_1}}, ..., \triangle_{stp_{i_n}}] \centerdot$
$[((routetoDes_{1\lhd m})^k \centerdot (gather\&barrier_{x\rhd 1}))_{i_1}, ...,$
$((routetoDes_{1\lhd m})^k \centerdot (gather\&barrier_{x\rhd 1}))_{i_n}]$

We now demonstrate that the *BSP* pattern can be captured by the proposed grammar.

$\square_{BSP} ::= \blacktriangle_{BSP}$

$\blacktriangle_{BSP} ::= \triangle_{BSP}$

$\triangle_{BSP} ::= \dashv^{1n} \centerdot \pentagon^n \centerdot \vdash^{n1}$

$\pentagon^n ::= (\blacktriangle)^n \centerdot \pentagon^n$

The last line is used to generate the p super-steps. The $(\blacktriangle)^n$ in the last line represents the first super-step. To generate the $p-1$ remaining super-steps, first, for $p-2$ times the right $\pentagon$ in the last line is replaced with $(\blacktriangle)^n \centerdot \pentagon^n$ recursively. Then, the right $\pentagon$ at the right side of the instruction is replaced with the $(\blacktriangle)^n$ to generate the last super-step. We represent a super-step i with ${(\blacktriangle)^n}_i$ where $i \in [1, ..., p-1]$. Therefore, a super-step i (${(\blacktriangle)^n}_i$) is captured by the proposed grammar as follows.

${(\blacktriangle)^n}_i ::= {(\blacktriangle)^n}_{i_{stp}} \centerdot {\pentagon^n}_{i_{reroute}}$

$\blacktriangle_{i_{stp}} ::= \triangle_{i_{stp}}$

$(\triangle_{i_{stp}})^n ::= [\triangle_{i_{stp}1}, ..., \triangle_{i_{stp}n}]$

$$\pentagon^{n}{}_{i_{reroute}} ::= (\blacktriangle_{i_{reroute}})^{n}$$
$$\blacktriangle_{i_{reroute}} ::= \triangle_{i_{reroute}}$$
$$(\triangle_{i_{reroute}})^{n} ::= [\triangle_{i_{reroute}1}, ..., \triangle_{i_{reroute}n}]$$
$$\triangle_{i_{reroute}j} ::= \dashv^{1x} \centerdot \vdash^{x1}$$
$$\dashv^{1x} ::= \dashv^{1x}_{mk}$$
$$\dashv^{1x}_{mk} ::= (\dashv^{1m})^{k}$$
$$\dashv^{1m} ::= (f_{RouttoDest1 \lhd m})$$
$$\vdash^{x1} ::= (gather\&barrier_{x \rhd 1})$$
$$(\blacktriangle^{n})_{p_{stp}} ::= (\triangle_{p_{stp}})^{n}$$
$$(\triangle_{p_{stp}})^{n} ::= [\triangle_{p_{stp}1}, ..., \triangle_{p_{stp}n}]$$
$$\dashv^{1n} ::= broadcast_{1 \lhd n}$$
$$\vdash^{n1} ::= gatherall_{n \rhd 1}$$

A.2.2 *Map-Reduce*

The *Map-Reduce* pattern, introduced by Google, models those applications for which a collection of input data is processed into the following three steps [121, 122]:

- A *Map* step: Computes a $(key : value)$ pair for each item in the collection by applying a function f;
- An intermediate processig step: Sorts and rearranges data according to their keys where pairs with the same keys are put into a single collection; and
- A *reduce* step: "Sums-up" all the value items for a given key using an associative and communicative reduction pattern $\triangle_{\oplus}$, represented in A.1.2.

Applying our grammar, the *Map-Reduce* pattern can be generated as follows.

$$\square_{mr} ::= \blacklozenge_{mr}$$
$$\blacklozenge_{mr} ::= \diamond_{mr}$$
$$\diamond_{mr} ::= \dashv^{1n} \centerdot \bigcirc^{n}$$
$$\bigcirc^{n}{}_{mr} ::= \pentagon^{n}{}_{map} \centerdot (\dashv^{1z})^{n} \centerdot (\vdash^{n1})^{z} \centerdot \bigcirc^{z}{}_{reduce}$$
$$\pentagon^{nmap} ::= (\blacktriangle_{map})^{n}$$
$$\blacktriangle_{map} ::= \triangle_{map}$$
$$(\triangle_{map})^{n} ::= [\triangle_{map}]_{n}$$
$$\bigcirc^{z}{}_{reduce} ::= \pentagon^{z}{}_{reduce}$$
$$\pentagon^{z}_{reduce} ::= (\blacktriangle_{\oplus})^{z}$$
$$\blacktriangle_{\oplus} ::= \triangle_{\oplus}$$
$$(\triangle_{\oplus})^{z} ::= [\triangle_{\oplus}]_{z}$$
$$\dashv^{1z} ::= (k_{1 \lhd z})$$

$\vdash^{n1} ::= gather_{n\triangleright 1}$

The instruction $(\dashv^{1z})^n \,.\, (\vdash^{n1})^z$ in line four represents the intermediate step. It redistributes and rearranges the data according to the number of *key*, where z represents the number of *key*. The number of components calculating the reduce step (z), can be refined by replacing $(\vdash^{n1})^z$ with $\vdash^{x1}_{mk} ::= (\vdash^{m1})^k . \vdash^{k1}$.

A.2.3 MDF

Using the functional dependencies among data, a program execution is a traversal of a graph whose nodes are *macro-dataflow* (*MDF*) instructions. Each MDF node becomes executable as soon as the input data is available on its input arcs. In [15] this model has been demonstrated as 3-stage pipeline operating for n iterations (where n represent the number of instructions), as follows:

- A function b to build the graph (possibly dynamically).
- A function e to determine the executable instruction.
- A function f to execute the MDF instruction in parallel and generate a token required for determining the next executable instruction.

The MDF graph can be constructed statically or dynamically. Depending on the type of constructing a MDF graph, two patterns can be considered for MDF [15]. In the following we explain each pattern.

A.2.3.1 Static MDF

The static MDF pattern is applied when the construction of the instruction graph is static. Using the RISC-pb^{2}l building blocks, the static MDF pattern can be represented as:

$\overleftarrow{\text{«b»} . (gather_{n\triangleright 1} . \text{«e»} . unicast_{1\triangleleft n} . [\text{«f»}]_n)_c}$

We now demonstrate that the above pattern is detectable by our grammar.

$\square_{MDF_S} ::= \blacklozenge_b$

$\blacklozenge_b ::= \blacktriangle_b . \blacklozenge_{ef}$

$\blacktriangle_b ::= \triangle_b$

$\triangle_b ::= \text{«b»}$

$\blacklozenge_{ef} ::= \overleftarrow{\Diamond_{ef}}$

$\Diamond_{ef} ::= (\models^{n1}_1 . \blacktriangle_e . \dashv^{1n} . \pentagon^n{}_f)_c$

$\blacktriangle_e ::= \triangle_e$

$\triangle_e ::= \text{«e»}$

$\pentagon^n{}_f ::= (\blacktriangle_f)^n$

$\blacktriangle_f := \triangle_f$

$(\triangle_f)^n ::= [\triangle_f]_n$

$\triangle_f ::= \text{«f»}$

$\dashv^{1n} ::= unicast_{1 \lhd n}$

$\models_1^{n1} ::= gather_{n \rhd 1}$

A.2.3.2 Dynamic MDF

The dynamic MDF pattern is applied when the construction of the instruction graph is dynamic. Using the RISC-pb²l building blocks, the dynamic MDF composition can be represented as:

$\overleftarrow{(gather_{n \rhd 1} \,.\, \text{«b»} \,.\, \text{«e»} \,.\, unicast_{1 \lhd n} \,.\, [\text{«f»}]_n)_c}$

We now demonstrate that the above pattern is detectable by the proposed grammar.

$\square_{MDF_S} ::= \blacklozenge_{bef}$

$\blacklozenge_{bef} ::= \Diamond_{bef}$

$\Diamond_{ef} ::= \overleftarrow{(\models_1^{n1} \,.\, \blacktriangle_{be} \,.\, \dashv^{1n} \,.\, \pentagon^n{}_f)_c}$

$\blacktriangle_{be} ::= \nabla_{be}$

$\nabla_{be} ::= \triangle_b \,.\, \triangle_e$

$\triangle_b ::= \text{«b»}$

$\triangle_e ::= \text{«e»}$

$\pentagon^n{}_f ::= (\blacktriangle_f)^n$

$\blacktriangle_f := \triangle_f$

$(\triangle_f)^n ::= [\triangle_f]_n$

$\triangle_f ::= \text{«f»}$

$\dashv^{1n} ::= unicast_{1 \lhd n}$

$\models_1^{n1} ::= gather_{n \rhd 1}$

A.3 Domain Specific pattern

In this section, we demonstrate that the three domain specific patterns *GSP*, *OB* and *NPP* generated by RISC-pb²l building block [15], can be captured by our grammar.

A.3.1 GSP

The global single population (GSP) genetic skeleton pattern belongs to the family of so-called genetic algorithm skeletons [123, 124].

Using the RISC-pb²l building block, the GSP pattern can be captured as follows [15]:

$\overleftarrow{(scatter_{1 \lhd n} \,.\, [\text{«eval»}] \,.\, (filter_{n \rhd 1}))_{term}}$

In the following we demonstrate that this pattern is detectable by the proposed grammar.

$\square_{gsp} ::= \blacktriangle_{gsp}$

$\blacktriangle_{gsp} ::= \triangle gsp$

$\triangle_{gsp} ::= \overleftarrow{(\ \dashv^{1n} . \pentagon^{n}_{gsp} . \vdash^{n1})}_{term}$

$\pentagon^{n}_{gsp} ::= (\blacktriangle_{gsp})^{n}$

$\blacktriangle_{gsp} := \triangle_{gsp}$

$(\triangle_{gsp})^{n} ::= [\triangle_{gsp}]_{n}$

$\triangle_{gsp} ::= \text{«eval»}$

$\dashv^{1n} ::= scatter_{1 \lhd n}$

$\vdash^{n1} ::= (filter_{n \rhd 1})$

A.3.2 OB

The Orbit Skeleton (OB) [125] belongs to the field of symbolic computation. Starting from an initial set of "points", it recursively applies a set of *generators* and adds those generated points that are not already included in the original set. The process will be repeated until the transitive closure of the *generators* on the initial set is eventually computed.

Using the RISC-pb^{2}l building block, the OB pattern can be captured as follows [15]:

$\overleftarrow{scatter_{1 \lhd n} . [brodcast_{1 \lhd x} . [\text{«}g_1\text{»}, \ldots, \text{«}g_x\text{»}] . (fD)_{x \rhd 1}]_n . (fD)_{n \rhd 1} . \text{«uS»}_{nia}}$

In the following we demonstrate that our grammar can detect the *OB* pattern.

$\square_{ob} ::= \blacktriangle_{o}$

$\blacktriangle_{o} ::= \triangle_{o}$

$\triangle_{o} ::= \overleftarrow{(\nabla_{o})}_{nia}$

$\nabla_{o} ::= \triangle_{ob} . \triangle_{uS}$

$\triangle_{ob} ::= \dashv^{1n} . \pentagon^{n}_{ob} . \vdash^{n1}$

$\pentagon^{n}_{ob} ::= (\blacktriangle_{obs})^{n}$

$\blacktriangle_{obs} ::= \triangle_{obs}$

$(\triangle_{obs})^{n} ::= [\triangle_{obs}]_{n}$

$\triangle_{obs} ::= \dashv^{1x} . \pentagon^{x}_{obs} . \vdash^{x1}$

$\pentagon^{x}_{obs} ::= (\blacktriangle_{obg})^{x}$

$\blacktriangle_{obg} ::= \triangle_{obg}$

$(\triangle_{obg})^{x} ::= [\triangle_{obg_1}, \ldots \triangle_{obg_x}]$

$\triangle_{obg_i} ::= \text{«}g_i\text{»}$

$\dashv^{1n} ::= scatter_{1 \lhd n}$

$\vdash^{n1} ::= (fD_{n \rhd 1})$

$\dashv^{1x} ::= broadcast_{1 \lhd x}$

$\vdash^{x1} ::= (fD_{x \rhd 1})$

$\triangle_{uS} ::= \text{«}uS\text{»}$

A.3.3 NPP

The network packet processing (NPP) can be typically considered as a pipeline of parallel functional stages. The number of stages in the pipeline depends on the processing application. Without loss of generality, a two stage NPP pipeline consists of the following stages:

- Parsing stage: To parse packets.
- Processing stage: To provide a protocol management and to process network packet-s/flows.

Using the RISC-pb^{2}l building blocks, the NPP pattern can be captured as follows [15]:

$(f_{1\lhd n}) \centerdot [\triangle_{pars}]_n \centerdot (f^{-1}_{n\rhd 1}) \centerdot (h_{1\lhd n}) \centerdot [\triangle_{proc}]_n \centerdot gather_{n\rhd 1}$

In the following we demonstrate that the proposed grammar can capture the NPP pattern.

$\square_{npp} ::= \blacktriangle_{npp}$

$\blacktriangle_{npp} ::= \triangledown_{npp}$

$\triangledown_{npp} ::= \triangle_{pars} \centerdot \triangle_{proc}$

$\triangle_{pars} ::= \dashv_{1n} \centerdot \pentagon^{n}_{pars} \centerdot \vdash^{n1}$

$\pentagon^{n}_{pars} ::= (\blacktriangle_{parsi})^n$

$\blacktriangle_{parsi} ::= \triangle_{parsi}$

$(\triangle_{parsi})^n ::= [\triangle_{parsi}]_n$

$\dashv^{1n} ::= (f_{1\lhd n})$

$\vdash^{n1} ::= (f^{-1}_{n\rhd 1})$

$\triangle_{proc} ::= \dashv_{1x} \centerdot \pentagon^{x}_{proci} \centerdot \vdash^{x1}$

$\pentagon^{x}_{proci} ::= (\blacktriangle_{proci})^x$

$\blacktriangle_{proci} ::= \triangle_{proci}$

$(\triangle_{proci})^x ::= [\triangle_{proci}]_x$

$\dashv^{1x} ::= (h_{1\lhd x})$

$\vdash^{x1} ::= gather_{x\rhd 1}$

Appendix B

The Structural Representation of Application Suite

B.1 Uniform Random Noise Generator

The URNG application has been designed as follow:

«readStage » . «||urng||» . «writeStage »

B.1.1 Demonstration of URNG with RISC-pb^{2}l Grammar

Using the building block grammar represented in Listing 3.1, the URNG application is generated as follows:

$\square_{URNG}$ ::= $\blacktriangle_{RUW}$

$\blacktriangle_{RUW}$ ::= $\triangledown_{RUW}$

$\triangledown_{RUW}$ ::= $\triangle_{R}$. $\triangledown_{UW}$

$\triangledown_{UW}$ ::= $\triangle_{U}$. $\triangle_{W}$

$\triangle_{R}$ ::= «readStage »

$\triangle_{U}$ ::= «||urng||»

$\triangle_{W}$ ::= «writeStage »

B.1.2 SKIP-compliant Object Representing the URNG Application

The following is a SKIP object representing the structural meta-data required to generate the URNG application.

```
{
  "Application_Name" : "URNG",
  "Profile_and_Tune" :true,
  "Priority" : 0,
  "Sampling_Mode":"Sparse",
```

```
  "ReductionPipeComposition":{
    "Component_Name" : "ff_pipeline",
    "Component_Type" :"pipeline",
    "Ch_Bound" : 200,
    "Sequential" :{
        "Component_Name" : "ff_node",
        "Component_Type" : "sequential",
        "Function_Name"  : "read_benchmarck"
    },
    "Hsequential" :{
        "Component_Name" : "oclnode",
        "Component_Type" : "hsequential",
        "Host_Preparation_Function": "set_oclParameter_URNG",
        "kernel_Path": "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/
          examples/URNG/URNG_Kernels.cl",
        "Function_Name"  : "execute_kernel_URNG"
    },
    "Sequential" :{
        "Component_Name" : "ff_node",
        "Component_Type" : "sequential",
        "Function_Name"  : "write_benchmarck"
    }
  }

}
```

B.2 Recursive Gaussian

The recursive Gaussian application has been designed as follow: «readStage » . «||gausianFilter||» . «||transpose||» . «||gausianFilter||» . «||transpose||» . «adjust » . «writeStage »

B.2.1 Demonstration of Recursive Gaussian with RISC-pb^{2}l Grammar

Using the building block grammar represented in Listing 3.1, the recursive Gaussian application is generated as follows:

$\square_{RG} ::= \blacktriangle_{RG1T1G2T2AW}$

$\blacktriangle_{RG1T1G2T2AW} ::= \nabla_{RG1T1G2T2AW}$

$\nabla_{RG1T1G2T2AW} ::= \triangle_{R} \cdot \nabla_{G1T1G2T2AW}$

$\nabla_{G1T1G2T2AW} ::= \triangle_{G1} \cdot \nabla_{T1G2T2AW}$

$\nabla_{T1G2T2AW} ::= \triangle_{T1} \cdot \nabla_{G2T2AW}$

$\nabla_{G2T2AW} ::= \triangle_{G2} \cdot \nabla_{T2AW}$

$\nabla_{T2AW} ::= \triangle_{T2} \cdot \nabla_{AW}$

$\nabla_{T2AW} ::= \triangle_{A} \cdot \triangle_{W}$

$\triangle_{R} ::=$ «readStage »

$\triangle_{G1}$ ::= «||gausianFilter||»

$\triangle_{T1}$ ::= «||transpose||»

$\triangle_{G2}$ ::= «||gausianFilter||»

$\triangle_{T2}$ ::= «||transpose||»

$\triangle_{A}$ ::= «adjust »

$\triangle_{W}$ ::= «writeStage »

B.2.2 SKIP-compliant Object Representing the Recursive Gaussian Application

The following is a SKIP object representing the structural meta-data required to generate the recursive Gaussian application.

```
{
  "Application_Name" : "recursive-gaussian",
  "Profile_and_Tune" : true,
  "Priority" : 0,
  "Sampling_Mode":"Sparse",
  "ReductionPipeComposition":{
    "Component_Name" :"ff_pipeline",
    "Component_Type" : "pipeline",
    "Ch_Bound" : 320,
    "Sequential" :{
        "Component_Name" :"ff_node",
        "Component_Type" : "sequential",
        "Function_Name" : "read_benchmarck"
    },
    "Hsequential" :{
        "Component_Name" :"oclnode",
        "Component_Type" : "hsequential",
        "Host_Preparation_Function" : "set_oclParameter_recursiveGaussian",
        "Kernel_Path" : "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/examples
            /RecursiveGaussian/RecursiveGaussian_Kernels.cl",
        "Function_Name"  : "execute_kernel_recursiveGaussian"
    },
    "Hsequential" :{
        "Component_Name" :"oclnode",
        "Component_Type" : "hsequential",
        "Host_Preparation_Function": "set_oclParameter_recursiveGaussian",
        "Kernel_Path" : "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/examples
             /RecursiveGaussian/RecursiveGaussian_Kernels.cl",
        "Function_Name"  : "execute_kernel_transpose"
    },
    "Hsequential" :{
        "Component_Name" :"oclnode",
        "Component_Type" : "hsequential",
        "Host_Preparation_Function" : "set_oclParameter_recursiveGaussian",
```

```
35          "Kernel_Path" : "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/examples
36              /RecursiveGaussian/RecursiveGaussian_Kernels.cl",
37          "Function_Name"  : "execute_kernel_recursiveGaussian"
38      },
39      "Hsequential" :{
40          "Component_Name" :"oclnode",
41          "Component_Type" : "hsequential",
42          "Host_Preparation_Function" : "set_oclParameter_recursiveGaussian",
43          "Kernel_Path" : "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/examples
44              /RecursiveGaussian/RecursiveGaussian_Kernels.cl",
45          "Function_Name"  : "execute_kernel_transpose"
46      },
47      "Sequential" :{
48          "Component_Name" :"ff_node",
49          "Component_Type" : "sequential",
50          "Function_Name"  : "adjust_result"
51      },
52      "Sequential" :{
53          "Component_Name" :"ff_node",
54          "Component_Type" : "sequential",
55          "Function_Name"  : "write_benchmarck"
56      }
57    }
58  }
```

B.3 Separable Convolution

The separable convolution application has been designed as follow:

«generation » . $Unicast_{1 \lhd n}$. $[$«‖convolutionFilter‖»$]_n$

B.3.1 Demonstration of Separable Convolution with RISC-pb^{2}l Grammar

Using the building block grammar represented in Listing 3.1, the separable convolution application can be demonstrated as:

$\square_{CF} ::= \blacklozenge_{GC}$

$\blacklozenge_{GC} ::= \blacktriangle_G \,.\, \blacklozenge_C$

$\blacktriangle_G ::= \triangle_G$

$\triangle_G ::=$ «generation »

$\blacklozenge_C ::= \diamondsuit_C$

$\diamondsuit_C ::= \dashv^{1n} . \bigcirc_C^n$

$\dashv^{1n} ::= Unicast_{1 \lhd n}$

$\bigcirc_C^n ::= \pentagon_C^n$

$\pentagon_C^n ::= (\blacktriangle_C)^n$

$(\blacktriangle_C)^n ::= [\blacktriangle_C]_n$

$\blacktriangle_C ::= \triangle_C$

$\triangle_C ::= \text{«||convolutionFilter||»}$

B.3.2 SKIP-compliant Object Representing the Separable Convolution Application

The following is a SKIP object representing the structural meta-data required to generate the separable convolution application.

```
{
    "Application_Name" : "separable-convolution",
    "Profile_and_Tune" :true,
    "Priority" : 0,
     "Sampling_Mode":"Sparse",
    "NonReductionPipeComposition":{
        "Component_Name" : "ff_pipeline",
        "Component_Type" : "pipeline",
        "Ch_Bound" :100,
        "Sequential" :{
                "Component_Name" :"ff_node",
                "Component_Type" : "Sequential",
                "Function_Name" : "first_stage_worker_function"
        },
        "NonReductionComposition":{
                "Component_Name"  :"ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>",
                "Component_Type" : "farm",
                "Ch_Bound" :100,
                "1ToNCom" : {
                      "Component_Name"  :"ff_loadbalancer",
                      "Component_Type" : "spread",
                      "Ch_Policy"      : "adaptive_loadbalancer",
                      "Workload"       : [1,1],
                      "Ch_Multiplicity": 2,
                      "Filter" :{
                            "Component_Name"  :"emitter",
                            "Component_Type" : "Sequential",
                            "Function_Name" : "iterative_stage_emitter_function"
                       }
                },
                "Parallel" : [
                    {
                        "Par_Level": 2,
                        "Hsequential" : {
                                "Component_Name"  :"oclnode",
                                "Component_Type" : "Hsequential",
                                "Host_Preparation_Function": "set_oclParameter",
                                "kernel_path": "/users/mehdi/Thesis/
          fastflow-2.0.2.13-VIP/examples/simple-convolution/
          Lib_version/SimpleConvolution_Kernels.cl",
```

```
                                "Function_Name" : "execute_kernel"
                            }
                        }
                    ]
            }
        }
}
```

B.4 Bilateral Denoise

B.4.1 Demonstration of Bilateral Denoise with RISC-pb^{2}l Grammar

The bilateral denoise application has been designed as follow:

«readStage » . $Unicast_{1 \lhd n}$. $[$«||bilateralDenoise||»$]_n$. $gather_{n \rhd 1}$. «writeStage »

Using the building block grammar represented in Listing 3.1, the bilateral denoise application is generated as follows:

$\square_{BD} ::= \blacktriangle_{RBW}$

$\blacktriangle_{RBW} ::= \triangledown_{RBW}$

$\triangledown_{RBW} ::= \triangle_R \cdot \triangledown_{BW}$

$\triangledown_{BW} ::= \triangle_B \cdot \triangle_W$

$\triangle_R ::=$ «readStage »

$\triangle_B ::= \dashv^{1n} . \pentagon_B^n \vdash^{n1}$

$\dashv^{1n} ::= Unicast_{1 \lhd n}$

$\pentagon_B^n ::= (\blacktriangle_B)^n$

$(\blacktriangle_B)^n ::= [\blacktriangle_B]_n$

$\blacktriangle_B ::= \triangle_B$

$\triangle_B ::=$ «||bilateralDenoise||»

$\vdash^{n1} ::= gather_{n \rhd 1}$

$\triangle_W ::=$ «writeStage »

B.4.2 SKIP-compliant Object Representing the Bilateral Denoise Application

The following is a SKIP compliant object representing the structural meta-data required to generate the bilateral denoise application.

```
{
  "Application_Name" : "bilateral-dnoise",
  "Profile_and_Tune" :true,
  "Priority" : 0,
  "Sampling_Mode":"Sparse",
```

```
"ReductionPipeComposition":{
  "Component_Name" : "ff_pipeline",
  "Component_Type" : "pipeline",
  "Ch_Bound" :200,
  "Sequential" :{
      "Component_Name" :"ff_node",
      "Component_Type" : "Sequential",
      "Function_Name"  : "read_benchmarck"
  },
  "ReductionComposition":{
      "Component_Name"  :"ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>",
      "Component_Type" : "farm",
      "Ch_Bound" :200,
      "1ToNCom" : {
          "Component_Name"  :"ff_loadbalancer",
          "Component_Type" : "spread",
          "Ch_Policy"      : "adaptive_loadbalancer",
          "Workload"       : [1,1],
          "Ch_Multiplicity": 2,
          "Filter" :{
            "Component_Name"  :"emitter",
            "Component_Type" : "Sequential",
              "Function_Name" : "iterative_stage_emitter_function"
           }
      },
      "Parallel" : [
        {
          "Par_Level": 2,
          "Hsequential" :  {
              "Component_Name"  :"oclnode",
              "Component_Type" : "Hsequential",
              "Host_Preparation_Function": "set_oclParameter_bilateral",
              "kernel_path": "/users/mehdi/Thesis/
                fastflow-2.0.2.13-VIP/examples/Bilateral-Denoise/bilateralKernel.cl",
              "Function_Name" : "execute_kernel_bilateral"
          }
        }
      ],
      "NTo1Com" : {
                "Component_Name"  :"collector",
              "Component_Type" : "Reduce",
              "Ch_Policy"      : "ff::ff_gatherer",
              "Ch_Multiplicity": 2,
              "Filter" :{
              "Component_Type" : "Sequential",
                "Function_Name" : "iterative_stage_collector_function"
              }
        }
  },
  "Sequential":  {
      "Component_Name"  :"ff_node",
```

```
            "Component_Type" : "Sequential",
          "Function_Name"   : "write_benchmarck"
      }
    }
}
```

B.5 Sobel Filter

The sobel filter application has been designed as follow:

«readStage » . «||sobelOperator||» . «writeStage »

B.5.1 Demonstration of Soble Filter with RISC-pb^{2}l Grammar

Using the building block grammar represented in Listing 3.1, the sobel filter application is generated as follows:

$\square_{SF} ::= \blacktriangle_{RSW}$

$\blacktriangle_{RSW} ::= \triangledown_{RSW}$

$\triangledown_{RSW} ::= \triangle_{R} \cdot \triangledown_{SW}$

$\triangledown_{SW} ::= \triangle_{S} \cdot \triangle_{W}$

$\triangle_{R} ::=$ «readStage »

$\triangle_{S} ::=$ «||sobelOperator||»

$\triangle_{W} ::=$ «writeStage »

B.5.2 SKIP-compliant Object Representing the Soble Filter Application

The following is a SKIP object representing the structural meta-data required to generate the sobel filter application.

```
{
  "Application_Name" : "SobelFilter",
  "Profile_and_Tune" :true,
  "Priority" : 0,
  "Sampling_Mode":"Sparse",
  "ReductionPipeComposition":{
    "Component_Name" : "ff_pipeline",
    "Component_Type" :"pipeline",
    "Ch_Bound" : 200,
    "Sequential" :{
        "Component_Name" : "ff_node",
        "Component_Type" : "sequential",
        "Function_Name"  : "read_benchmarck"
    },
    "Hsequential" :{
```

```
        "Component_Name" : "oclnode",
        "Component_Type" : "hsequential",
        "Host_Preparation_Function": "set_oclParameter_sobel",
        "kernel_Path": "/users/mehdi/Thesis/fastflow-2.0.2.13-VIP/examples/
            SobelFilter/SobelFilter_Kernels.cl",
        "Function_Name"  : "execute_kernel_sobel"
    },
    "Sequential" :{
        "Component_Name" : "ff_node",
        "Component_Type" : "sequential",
        "Function_Name"  : "write_benchmarck"
    }
  }

}
```

B.6 Gaussian Noise

The Gaussian-Noise application has been designed as follow:

$(ReadUnicast_{1 \lhd n})$. $[\text{«‖gausianNoise‖»}]_n$. $(Writegather_{n \rhd 1})$

B.6.1 Demonstration of Gaussian Noise with RISC-pb^{2}l Grammar

Using the building block grammar represented in Listing 3.1, the Gaussian noise application is generated as follows:

$\square_{GN} ::= \blacktriangle_{RGW}$

$\blacktriangle_{RGW} ::= \triangle_{RGW}$

$\triangle_{RGW} ::= \dashv^{1n}$. $\pentagon_G^n \vdash^{n1}$

$\dashv^{1n} ::= (ReadUnicast_{1 \lhd n})$

$\pentagon_G^n ::= (\blacktriangle_G)^n$

$(\blacktriangle_G)^n ::= [\blacktriangle_G]_n$

$\blacktriangle_G ::= \triangle_G$

$\triangle_G ::= \text{«‖gausianNoise‖»}$

$\vdash^{n1} ::= (Writegather_{n \rhd 1})$

B.6.2 SKIP-compliant Object Representing the Gaussian Noise Application

The following is a SKIP object representing the structural meta-data required to generate the Gaussian noise application.

```
{
  "Application_Name" : "bilateral-dnoise",
  "Profile_and_Tune" :true,
```

```
  "Priority" : 0,
  "Sampling_Mode":"Sparse",

  "ReductionComposition":{
      "Component_Name"  :"ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>",
      "Component_Type" : "farm",
      "Ch_Bound" :200,
      "1ToNCom" : {
          "Component_Name"  :"ff_loadbalancer",
          "Component_Type" : "spread",
          "Ch_Policy"      : "adaptive_loadbalancer",
          "Workload"       : [1,1],
          "Ch_Multiplicity": 2,
          "Filter" :{
            "Component_Name"  :"emitter",
            "Component_Type" : "Sequential",
              "Function_Name" : "read_benchmarck"
           }
      },
      "Parallel" : [
        {
          "Par_Level": 2,
          "Hsequential" :  {
              "Component_Name"  :"oclnode",
              "Component_Type" : "Hsequential",
              "Host_Preparation_Function": "set_oclParameter_gaussian_transform",
              "kernel_path": "/users/mehdi/Thesis/
                fastflow-2.0.2.13-VIP/examples/Gausian-Noise/GaussianNoise_Kernels.cl",
              "Function_Name" : "execute_kernel_gaussian_transform"
          }
        }
      ],
      "NTo1Com" : {
                "Component_Name"  :"collector",
              "Component_Type" : "Reduce",
              "Ch_Policy"      : "ff::ff_gatherer",
              "Ch_Multiplicity": 2,
              "Filter" :{
              "Component_Type" : "Sequential",
                "Function_Name" : "write_benchmarck"
              }
          }
  }
}
```

Appendix C

The SKIP Compliant Objects

C.1 Sensor Files

In the following we demonstrate a snapshot of the sensor file for each application that uses our *HWrapper* building block. Each file has a tree structure. In all cases, the root of the tree has no name as the C++ JSON library applied here eliminates the name of the root for a JSON object.

C.1.1 Bilateral-Denoise


```
{
  "Component_Name" : "ff::ff_pipeline",
  "Component_Type" : "pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Elapsed_Time" : 870365.957,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6145,
    "Queue_Input" : -1,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 23231.93,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 10060213,
    "Push_Delay_Time" : 10060.213,
    "Pop_Delay_Count" : 0,
    "Pop_Delay_Time" : 0,
    "End_Received" : 0
  },
  "ReductionComposition" : {
    "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
```

```
    "Component_Type" : "farm",
    "Node_Address" : "\/ff_pipeline\/0\
      /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1",
    "1ToNCom" : {
      "Component_Name" : "ff::adaptive_loadbalancer",
      "Ch_Policy" : "adaptive_loadbalancer",
      "Ch_Out" : 6144,
      "Ch_In" : 6144,
      "Elapsed_Time" : 870365.993,
      "Component_Type" : "spread",
      "Node_Address" : "\/ff_pipeline\/0\
        /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/ff_loadbalancer\/0",
      "Assigned_Processor" : -1,
      "Sampling_Rate" : 0,
      "Filter" : {
        "Component_Name" : "emitter_cls",
        "Component_Type" : "ff_node",
        "Component_Time_Distribution" : []
      },
      "Component_Last_Processing_Time" : 0,
      "Total_Component_Active_Time" : 0,
      "Push_Delay_Count" : 12250980,
      "Push_Delay_Time" : 12250.98,
      "Pop_Delay_Count" : 586,
      "Pop_Delay_Time" : 586000
    },
    "parallel" : [
      {
        "Hsequential" : {
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Node_Address" : "\/ff_pipeline\/0\
            /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/0",
          "Elapsed_Time" : 870366.013,
          "Assigned_Processor" : -1,
          "Processed_Tasks" : 3709,
          "Queue_Input" : 0,
          "Queue_Output" : 0,
          "Component_Last_Processing_Time" : 0,
          "Total_Component_Active_Time" : 600604.333,
          "Sampling_Rate" : 0,
          "Component_Time_Distribution" : [],
          "Push_Delay_Count" : 0,
          "Push_Delay_Time" : 0,
          "Pop_Delay_Count" : 3013468,
          "Pop_Delay_Time" : 3013.468,
          "End_Received" : 0,
          "Assigned_Device_Number" : 1
        },
        "Hsequential" : {
          "Component_Name" : "ocl_component_cls",
```

```
            "Component_Type" : "hsequential",
            "Node_Address" : "\/ff_pipeline\/0\
              /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/1",
            "Elapsed_Time" : 870366.039,
            "Assigned_Processor" : -1,
            "Processed_Tasks" : 183,
            "Queue_Input" : 2,
            "Queue_Output" : 0,
            "Component_Last_Processing_Time" : 553.294,
            "Total_Component_Active_Time" : 100867.747,
            "Sampling_Rate" : 0,
            "Component_Time_Distribution" : [],
            "Push_Delay_Count" : 0,
            "Push_Delay_Time" : 0,
            "Pop_Delay_Count" : 57,
            "Pop_Delay_Time" : 0.057,
            "End_Received" : 0,
            "Assigned_Device_Number" : 1
          }
        }
      ],
      "NTo1Com" : {
        "Component_Name" : "ff::ff_gatherer",
        "Ch_Policy" : "ff_gatherer",
        "Processed_Tasks" : 6142,
        "Elapsed_Time" : 870366.062,
        "Queue_Output" : 0,
        "Component_Type" : "reduce",
        "Node_Address" : "\/ff_pipeline\/0\
          /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/collector\/0",
        "Assigned_Processor" : -1,
        "Sampling_Rate" : 0,
        "Filter" : {
          "Component_Name" : "collector_cls",
          "Component_Type" : "ff_node",
          "Component_Time_Distribution" : []
        },
        "Component_Last_Processing_Time" : 0.002,
        "Total_Component_Active_Time" : 3.426,
        "Push_Delay_Count" : 0,
        "Push_Delay_Time" : 0,
        "Pop_Delay_Count" : 15757570,
        "Pop_Delay_Time" : 78787.85
      },
      "Elapsed_Time" : 870366.075
    },
    "Sequential" : {
      "Component_Name" : "component_cls",
      "Component_Type" : "sequential",
      "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
      "Elapsed_Time" : 870366.08,
```

```
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6142,
    "Queue_Input" : 0,
    "Queue_Output" : -1,
    "Component_Last_Processing_Time" : 0.01,
    "Total_Component_Active_Time" : 415.382,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 16083078,
    "Pop_Delay_Time" : 16083.078,
    "End_Received" : 0
  },
  "Elapsed_Time" : 870366.091
}
```


C.1.2 Recursive Gaussian


```
{
  "Component_Name" : "ff::ff_pipeline",
  "Component_Type" : "pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Elapsed_Time" : 109294.143,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6145,
    "Queue_Input" : -1,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 27679.278,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 526035,
    "Push_Delay_Time" : 526.035,
    "Pop_Delay_Count" : 0,
    "Pop_Delay_Time" : 0,
    "End_Received" : 0
  },
  "Hsequential" : {
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
    "Elapsed_Time" : 109294.176,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 128,
```

```
    "Queue_Input" : 0,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 3164.152,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 118727,
    "Push_Delay_Time" : 118.727,
    "Pop_Delay_Count" : 29,
    "Pop_Delay_Time" : 0.029,
    "End_Received" : 0,
    "Assigned_Device_Number" : 1
  },
  "Hsequential" : {
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/2",
    "Elapsed_Time" : 109294.2,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 5,
    "Queue_Input" : 0,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 237.286,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 416643,
    "Push_Delay_Time" : 416.643,
    "Pop_Delay_Count" : 5108,
    "Pop_Delay_Time" : 5.108,
    "End_Received" : 0,
    "Assigned_Device_Number" : 1
  },
  "Hsequential" : {
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/3",
    "Elapsed_Time" : 109294.225,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 332,
    "Queue_Input" : 0,
    "Queue_Output" : 5,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 12640.781,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 379220,
    "Push_Delay_Time" : 379.22,
    "Pop_Delay_Count" : 15247,
    "Pop_Delay_Time" : 15.247,
    "End_Received" : 0,
```

```
    "Assigned_Device_Number" : 1
  },
  "Hsequential" : {
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/4",
    "Elapsed_Time" : 109294.247,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 511,
    "Queue_Input" : 5,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 20.976,
    "Total_Component_Active_Time" : 16029.504,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 117703,
    "Pop_Delay_Time" : 117.703,
    "End_Received" : 0,
    "Assigned_Device_Number" : 1
  },
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/5",
    "Elapsed_Time" : 109294.265,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6139,
    "Queue_Input" : 0,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0.01,
    "Total_Component_Active_Time" : 28003.384,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 1489235,
    "Pop_Delay_Time" : 1489.235,
    "End_Received" : 0
  },
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/6",
    "Elapsed_Time" : 109294.282,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6138,
    "Queue_Input" : 0,
    "Queue_Output" : -1,
    "Component_Last_Processing_Time" : 0.01,
```

```
    "Total_Component_Active_Time" : 359.431,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 1998961,
    "Pop_Delay_Time" : 1998.961,
    "End_Received" : 0
  },
  "Elapsed_Time" : 109294.294
}
```


C.1.3 Gausian-Noise


```
{
  "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
  "Component_Type" : "farm",
  "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/0",
  "1ToNCom" : {
    "Component_Name" : "ff::adaptive_loadbalancer",
    "Ch_Policy" : "adaptive_loadbalancer",
    "Ch_Out" : 6144,
    "Ch_In" : 6144,
    "Elapsed_Time" : 68163.683,
    "Component_Type" : "spread",
    "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
      \/0\/ff_loadbalancer\/0",
    "Assigned_Processor" : -1,
    "Sampling_Rate" : 0,
    "Filter" : {
      "Component_Name" : "emitter_cls",
      "Component_Type" : "ff_node",
      "Component_Time_Distribution" : []
    },
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 0,
    "Push_Delay_Count" : 962364,
    "Push_Delay_Time" : 962.364,
    "Pop_Delay_Count" : 229,
    "Pop_Delay_Time" : 229000
  },
  "parallel" : [
    {
      "Hsequential" : {
        "Component_Name" : "ocl_component_cls",
        "Component_Type" : "hsequential",
        "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
          \/0\/oclnode\/0",
        "Elapsed_Time" : 68163.703,
```

```
        "Assigned_Processor" : -1,
        "Processed_Tasks" : 4467,
        "Queue_Input" : 0,
        "Queue_Output" : 0,
        "Component_Last_Processing_Time" : 0,
        "Total_Component_Active_Time" : 61034.073,
        "Sampling_Rate" : 0,
        "Component_Time_Distribution" : [],
        "Push_Delay_Count" : 0,
        "Push_Delay_Time" : 0,
        "Pop_Delay_Count" : 26,
        "Pop_Delay_Time" : 0.026,
        "End_Received" : 0,
        "Assigned_Device_Number" : 1
      },
      "Hsequential" : {
        "Component_Name" : "ocl_component_cls",
        "Component_Type" : "hsequential",
        "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
          \/0\/oclnode\/1",
        "Elapsed_Time" : 68163.721,
        "Assigned_Processor" : -1,
        "Processed_Tasks" : 158,
        "Queue_Input" : 11,
        "Queue_Output" : 0,
        "Component_Last_Processing_Time" : 48.345,
        "Total_Component_Active_Time" : 6937.662,
        "Sampling_Rate" : 0,
        "Component_Time_Distribution" : [],
        "Push_Delay_Count" : 0,
        "Push_Delay_Time" : 0,
        "Pop_Delay_Count" : 26,
        "Pop_Delay_Time" : 0.026,
        "End_Received" : 0,
        "Assigned_Device_Number" : 1
      }
    }
  ],
  "NTo1Com" : {
    "Component_Name" : "ff::ff_gatherer",
    "Ch_Policy" : "ff_gatherer",
    "Processed_Tasks" : 6133,
    "Elapsed_Time" : 68163.739,
    "Queue_Output" : 0,
    "Component_Type" : "reduce",
    "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
      \/0\/collector\/0",
    "Assigned_Processor" : -1,
    "Sampling_Rate" : 0,
    "Filter" : {
      "Component_Name" : "collector_cls",
```

```
      "Component_Type" : "ff_node",
      "Component_Time_Distribution" : []
    },
    "Component_Last_Processing_Time" : 0.001,
    "Total_Component_Active_Time" : 4.986,
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 1202015,
    "Pop_Delay_Time" : 6010.075
  },
  "Elapsed_Time" : 68163.754
}
```


C.1.4 Sobel Filter


```
{
  "Component_Name" : "ff::ff_pipeline",
  "Component_Type" : "pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Elapsed_Time" : 128351.531,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6145,
    "Queue_Input" : -1,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 19646.577,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 1645559,
    "Push_Delay_Time" : 1645.559,
    "Pop_Delay_Count" : 0,
    "Pop_Delay_Time" : 0,
    "End_Received" : 0
  },
  "Hsequential" : {
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
    "Elapsed_Time" : 128351.565,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6144,
    "Queue_Input" : 0,
    "Queue_Output" : 1,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 128300.18,
```

```
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 33,
    "Pop_Delay_Time" : 0.033,
    "End_Received" : 0,
    "Assigned_Device_Number" : 1
  },
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
    "Elapsed_Time" : 128351.586,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6144,
    "Queue_Input" : 1,
    "Queue_Output" : -1,
    "Component_Last_Processing_Time" : 0.01,
    "Total_Component_Active_Time" : 291.969,
    "Sampling_Rate" : 0,
    "Component_Time_Distribution" : [],
    "Push_Delay_Count" : 0,
    "Push_Delay_Time" : 0,
    "Pop_Delay_Count" : 2326091,
    "Pop_Delay_Time" : 2326.091,
    "End_Received" : 0
  },
  "Elapsed_Time" : 128351.599
}
```


C.1.5 separable-Convolution


```
{
  "Component_Name" : "ff::ff_pipeline",
  "Component_Type" : "pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Elapsed_Time" : 770831.917,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6145,
    "Queue_Input" : -1,
    "Queue_Output" : 0,
    "Component_Last_Processing_Time" : 0,
    "Total_Component_Active_Time" : 12063.308,
    "Sampling_Rate" : 0,
```

```
      "Component_Time_Distribution" : [],
      "Push_Delay_Count" : 12984490,
      "Push_Delay_Time" : 12984.49,
      "Pop_Delay_Count" : 0,
      "Pop_Delay_Time" : 0,
      "End_Received" : 0
    },
    "NonReductionComposition" : {
      "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
      "Component_Type" : "farm",
      "Node_Address" : "\/ff_pipeline\/0\
          /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1",
      "1ToNCom" : {
        "Component_Name" : "ff::adaptive_loadbalancer",
        "Ch_Policy" : "adaptive_loadbalancer",
        "Ch_Out" : 6144,
        "Ch_In" : 6144,
        "Elapsed_Time" : 770831.935,
        "Component_Type" : "spread",
        "Node_Address" : "\/ff_pipeline\/0\
          /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/ff_loadbalancer\/0",
        "Assigned_Processor" : -1,
        "Sampling_Rate" : 0,
        "Filter" : {
          "Component_Name" : "emitter_cls",
          "Component_Type" : "ff_node",
          "Component_Time_Distribution" : []
        },
        "Component_Last_Processing_Time" : 0,
        "Total_Component_Active_Time" : 0,
        "Push_Delay_Count" : 13230466,
        "Push_Delay_Time" : 13230.466,
        "Pop_Delay_Count" : 3479,
        "Pop_Delay_Time" : 3479000
      },
      "parallel" : [
        {
          "Hsequential" : {
            "Component_Name" : "ocl_component_cls",
            "Component_Type" : "hsequential",
            "Node_Address" : "\/ff_pipeline\/0\
              /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/0",
            "Elapsed_Time" : 770831.948,
            "Assigned_Processor" : -1,
            "Processed_Tasks" : 5003,
            "Queue_Input" : 0,
            "Queue_Output" : 0,
            "Component_Last_Processing_Time" : 0,
            "Total_Component_Active_Time" : 749924.713,
            "Sampling_Rate" : 0,
            "Component_Time_Distribution" : [],
```

```
            "Push_Delay_Count" : 0,
            "Push_Delay_Time" : 0,
            "Pop_Delay_Count" : 112,
            "Pop_Delay_Time" : 0.112,
            "End_Received" : 0,
            "Assigned_Device_Number" : 1
          },
          "Hsequential" : {
            "Component_Name" : "ocl_component_cls",
            "Component_Type" : "hsequential",
            "Node_Address" : "\/ff_pipeline\/0\
              /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/1",
            "Elapsed_Time" : 770831.974,
            "Assigned_Processor" : -1,
            "Processed_Tasks" : 76,
            "Queue_Input" : 3,
            "Queue_Output" : 0,
            "Component_Last_Processing_Time" : 259.213,
            "Total_Component_Active_Time" : 19434.203,
            "Sampling_Rate" : 0,
            "Component_Time_Distribution" : [],
            "Push_Delay_Count" : 0,
            "Push_Delay_Time" : 0,
            "Pop_Delay_Count" : 129,
            "Pop_Delay_Time" : 0.129,
            "End_Received" : 0,
            "Assigned_Device_Number" : 1
          }
        }
      ],
      "Elapsed_Time" : 770831.998
    },
    "Elapsed_Time" : 770832.01
}
```


C.1.6 URNG


```
{
  "Component_Name" : "ff::ff_pipeline",
  "Component_Type" : "pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Sequential" : {
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Elapsed_Time" : 164249.834,
    "Assigned_Processor" : -1,
    "Processed_Tasks" : 6145,
    "Queue_Input" : -1,
```

```
      "Queue_Output" : 7,
      "Component_Last_Processing_Time" : 0,
      "Total_Component_Active_Time" : 20399.789,
      "Sampling_Rate" : 0,
      "Component_Time_Distribution" : [],
      "Push_Delay_Count" : 2260003,
      "Push_Delay_Time" : 2260.003,
      "Pop_Delay_Count" : 0,
      "Pop_Delay_Time" : 0,
      "End_Received" : 0
    },
    "Hsequential" : {
      "Component_Name" : "ocl_component_cls",
      "Component_Type" : "hsequential",
      "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
      "Elapsed_Time" : 164249.858,
      "Assigned_Processor" : -1,
      "Processed_Tasks" : 6138,
      "Queue_Input" : 7,
      "Queue_Output" : 0,
      "Component_Last_Processing_Time" : 112.546,
      "Total_Component_Active_Time" : 164156.688,
      "Sampling_Rate" : 0,
      "Component_Time_Distribution" : [],
      "Push_Delay_Count" : 0,
      "Push_Delay_Time" : 0,
      "Pop_Delay_Count" : 30,
      "Pop_Delay_Time" : 0.03,
      "End_Received" : 0,
      "Assigned_Device_Number" : 1
    },
    "Sequential" : {
      "Component_Name" : "component_cls",
      "Component_Type" : "sequential",
      "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
      "Elapsed_Time" : 164249.881,
      "Assigned_Processor" : -1,
      "Processed_Tasks" : 6137,
      "Queue_Input" : 0,
      "Queue_Output" : -1,
      "Component_Last_Processing_Time" : 0.01,
      "Total_Component_Active_Time" : 231.166,
      "Sampling_Rate" : 0,
      "Component_Time_Distribution" : [],
      "Push_Delay_Count" : 0,
      "Push_Delay_Time" : 0,
      "Pop_Delay_Count" : 3002733,
      "Pop_Delay_Time" : 3002.733,
      "End_Received" : 0
    },
    "Elapsed_Time" : 164249.893
```

```
}
```

C.2 Actuator Files

In the following we demonstrate a snapshot of the actuator file for each application that uses our *HWrapper* building block. Each file has a tree structure. In all cases, the root of the tree has no name as the C++ JSON library applied here eliminates the name of the root for a JSON object.

C.2.1 Bilateral-Denoise

```
{
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "ReductionComposition" : {
    "Node_Address" : "\/ff_pipeline\/0\
      /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1",
    "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
    "Component_Type" : "farm",
    "1ToNCom" : {
      "Node_Address" : "\/ff_pipeline\/0\
        /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/ff_loadbalancer\/0",
      "Component_Name" : "ff::adaptive_loadbalancer",
      "Component_Type" : "spread",
      "Assigned_Processor" : 3,
      "Filter" : {
        "Component_Name" : "emitter_cls"
      },
      "Masking_Array" : [
        1,
        1
      ],
      "Workload" : [
        0.79209,
        0.20791
      ]
    },
    "Parallel" : [
      {
        "Hsequential" : {
          "Node_Address" : "\/ff_pipeline\/0\
            /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/0",
          "Component_Name" : "ocl_component_cls",
```

```
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : 1
        },
        "Hsequential" : {
          "Node_Address" : "\/ff_pipeline\/0\
            /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/1",
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : "1"
        }
      }
    ],
    "Nto1Com" : {
      "Node_Address" : "\/ff_pipeline\/0\
        /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/collector\/0",
      "Component_Name" : "ff::ff_gatherer",
      "Component_Type" : "reduce",
      "Processed_Tasks" : 0,
      "Assigned_Processor" : 3,
      "Filter" : {
        "Component_Name" : "collector_cls"
      }
    }
  },
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "Component_Name" : "ff::ff_pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Component_Type" : "pipeline"
```

C.2.2 Gaussian-Noise

```
{
  "1ToNCom" : {
    "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
      \/0\/ff_loadbalancer\/0",
    "Component_Name" : "ff::adaptive_loadbalancer",
    "Component_Type" : "spread",
    "Assigned_Processor" : 3,
    "Filter" : {
      "Component_Name" : "emitter_cls"
    },
    "Masking_Array" : [
```

```
        1,
        1
      ],
      "Workload" : [
        0.76775,
        0.23225
      ]
    },
    "Parallel" : [
      {
        "Hsequential" : {
          "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
            \/0\/oclnode\/0",
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : 1
        },
        "Hsequential" : {
          "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
            \/0\/oclnode\/1",
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : "1"
        }
      }
    ],
    "Nto1Com" : {
      "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>
        \/0\/collector\/0",
      "Component_Name" : "ff::ff_gatherer",
      "Component_Type" : "reduce",
      "Processed_Tasks" : 0,
      "Assigned_Processor" : 3,
      "Filter" : {
        "Component_Name" : "collector_cls"
      }
    },
    "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
    "Node_Address" : "\/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/0",
    "Component_Type" : "farm",
}
```


C.2.3 Recursive-Gaussian


```
{
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
```

```
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 3,
    "Device_Number" : 1
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/2",
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 3,
    "Device_Number" : 0
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/3",
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 3,
    "Device_Number" : 1
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/4",
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 3,
    "Device_Number" : 0
  },
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/5",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/6",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "Component_Name" : "ff::ff_pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Component_Type" : "pipeline"
}
```

C.2.4 Sobel-Filter

```
{
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 4
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
    "Component_Name" : "ocl_component_cls",
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 4,
    "Device_Number" : 1
  },
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 4
  },
  "Component_Name" : "ff::ff_pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Component_Type" : "pipeline"
}
```

C.2.5 Separable-Convolution

```
{
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 3
  },
  "NonReductionComposition" : {
    "Node_Address" : "\/ff_pipeline\/0
      \/ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1",
    "Component_Name" : "ff::ff_farm<ff::adaptive_loadbalancer, ff::ff_gatherer>",
    "Component_Type" : "farm",
    "1ToNCom" : {
      "Node_Address" : "\/ff_pipeline\/0\
        /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/ff_loadbalancer\/0",
      "Component_Name" : "ff::adaptive_loadbalancer",
      "Component_Type" : "spread",
      "Assigned_Processor" : 3,
      "Filter" : {
        "Component_Name" : "emitter_cls"
```

```
      },
      "Masking_Array" : [
        1,
        1
      ],
      "Workload" : [
        0.825078,
        0.174922
      ]
    },
    "Parallel" : [
      {
        "Hsequential" : {
          "Node_Address" : "\/ff_pipeline\/0\
            /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/0",
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : 1
        },
        "Hsequential" : {
          "Node_Address" : "\/ff_pipeline\/0\
            /ff::ff_farm<adaptive_loadbalancer, ff::ff_gatherer>\/1\/oclnode\/1",
          "Component_Name" : "ocl_component_cls",
          "Component_Type" : "hsequential",
          "Assigned_Processor" : 3,
          "Device_Number" : "1"
        }
      }
    ]
  },

  "Component_Name" : "ff::ff_pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Component_Type" : "pipeline"
}
```

C.2.6 URNG

```
{
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/0",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 4
  },
  "Hsequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/oclnode\/1",
    "Component_Name" : "ocl_component_cls",
```

```
    "Component_Type" : "hsequential",
    "Assigned_Processor" : 4,
    "Device_Number" : 1
  },
  "sequential" : {
    "Node_Address" : "\/ff_pipeline\/0\/ff_node\/2",
    "Component_Name" : "component_cls",
    "Component_Type" : "sequential",
    "Assigned_Processor" : 4
  },
  "Component_Name" : "ff::ff_pipeline",
  "Node_Address" : "\/ff_pipeline\/0",
  "Component_Type" : "pipeline"
}
```

C.3 Constraint

```
{
  "Device" :{
    "Name": "CPU",
    "Maximum_BB_Per_Device": 1
  },
  "Device" :{
    "Name": "GPU",
    "Maximum_BB_Per_Device": 1
  },
  "Damping_Ratio":1.0,
  "Priority_Policy":"variable",
  "Component_Switch_Mode":"average"
}
```

Appendix D

The Structural Representation of Existing Applications

D.1 N-body Simulation

The N-body Simulation has encapsulated three FastFlow farm patterns:

Momentum $\square_{Momentum}$ presents a composition calculating the initial momentum of the bodies in the system. The pattern has been design as follows:

$(f_{momentum_{1 \lhd n}}) \centerdot [\text{«momentom»}]_n \centerdot (g_{momentum_{n \rhd 1}})$

Using the building block grammar represented in listing 3.1, the pattern is generated as follows:

$\square_{Momentum} ::= \blacktriangle_M$

$\blacktriangle_M ::= \triangle_M$

$\triangle_M ::= \dashv^{1n} \centerdot \pentagon_M^n \vdash_{n1}$

$\dashv^{1n} ::= (f_{momentum_{1 \lhd n}})$

$\pentagon_M^n ::= (\blacktriangle_M)^n$

$(\blacktriangle_M)^n ::= [\blacktriangle_M]_n$

$\blacktriangle_M ::= \triangle_M$

$\triangle_M ::= \text{«momentom»}$

$\vdash^{n1} ::= (g_{momentum_{n \rhd 1}})$

Energy $\square_{Energy}$ shows a composition calculating the total energy of the system before and after movement. The pattern has been design as follows:

$(f_{energy_{1 \lhd n}}) \centerdot [\text{«energy»}]_n \centerdot (g_{energy_{n \rhd 1}})$

Using the building block grammar represented in listing 3.1, the pattern is generated as follows:

$\square_{Energy} ::= \blacktriangle_E$

$\blacktriangle_E ::= \triangle_E$

$\triangle_E ::= \dashv^{1n} . \pentagon_E^n \vdash_{n1}$

$\dashv^{1n} ::= (f_{energy_{1 \lhd n}})$

$\pentagon_E^n ::= (\blacktriangle_E)^n$

$(\blacktriangle_E)^n ::= [\blacktriangle_E]_n$

$\blacktriangle_E ::= \triangle_E$

$\triangle_E ::=$ «energy»

$\vdash^{n1} ::= (g_{energy_{n \rhd 1}})$

Advance

$\square_{Advance}$ indicates a composition calculating the motion of the bodies towards the initial position after s step iterations. It has been encapsulated inside a `for loop` to calculate the motion of all bodies for s steps. The pattern has been design as follows:

$(f_{advance_{1 \lhd n}}) . [\text{«advance»}]_n . (g_{advance_{n \rhd 1}})$

Using the building block grammar represented in listing 3.1, the pattern is generated as follows:

$\square_{Advance} ::= \blacktriangle_A$

$\blacktriangle_A ::= \triangle_A$

$\triangle_A ::= \dashv^{1n} . \pentagon_A^n \vdash^{n1}$

$\dashv^{1n} ::= (f_{advance_{1 \lhd n}})$

$\pentagon_A^n ::= (\blacktriangle_A)^n$

$(\blacktriangle_A)^n ::= [\blacktriangle_A]_n$

$\blacktriangle_A ::= \triangle_A$

$\triangle_A ::=$ «advance»

$\vdash^{n1} ::= (g_{advance_{n \rhd 1}})$

D.2 Mandelbrot

The Mandelbrot application has been designed as follow:

$scatter_{1 \lhd n} . [\text{«mandeleq»}]_n . (gatherall\&drow_{n \rhd 1})$

Using the building block grammar represented in listing 3.1, the application is generated as follows:

$\square_{MB} ::= \blacktriangle_{MB}$

$\blacktriangle_{MB} ::= \triangle_{MB}$

$\triangle_{MB} ::= \dashv^{1n} . \pentagon^{n}_{MB} \vdash^{n1}$

$\dashv^{1n} ::= scatter_{1 \lhd n}$

$\pentagon^{n}_{MB} ::= (\blacktriangle_{MB})^{n}$

$(\blacktriangle_{MB})^{n} ::= [\blacktriangle_{MB}]_{n}$

$\blacktriangle_{MB} ::= \triangle_{MB}$

$\triangle_{MB} ::=$ «mandeleq»

$\vdash^{n1} ::= (gatherall\&drow_{n \rhd 1})$

D.3 Quick sort

The quick sort application has been designed as follow:

$\overleftarrow{((pivot_{dev1 \lhd n}) . [\text{«sort\_pivot »}]_{n} . (collect\&check_{n \rhd 1})_{termination}}$

Using the building block grammar represented in listing 3.1, the application is generated as follows:

$\square_{QS} ::= \blacktriangle_{QS}$

$\blacktriangle_{QS} ::= \triangle_{QS}$

$\triangle_{QS} ::= \overleftarrow{(\dashv^{1n} . \pentagon^{n}_{QS} \vdash^{n1})_{termination}}$

$\dashv^{1n} ::= (pivot_{dev1 \lhd n})$

$\pentagon^{n}_{QS} ::= (\blacktriangle_{QS})^{n}$

$(\blacktriangle_{QS})^{n} ::= [\blacktriangle_{QS}]_{n}$

$\blacktriangle_{QS} ::= \triangle_{QS}$

$\triangle_{QS} ::=$ «sort_pivot»

$\vdash^{n1} ::= (collect\&check_{n \rhd 1})$

D.4 Fibonacci

The Fibonacci application has been designed as follow:

$((thresholdBreaker_{1 \lhd n}) . [\text{«recursiveFib »}]_{n} . (\oplus_{n \rhd 1})$

Using the building block grammar represented in listing 3.1, the Fibonacci application is generated as follows:

$\square_{F} ::= \blacktriangle_{F}$

$\blacktriangle_{F} ::= \triangle_{F}$

$\triangle_{F} ::= \dashv^{1n} . \pentagon^{n}_{F} \vdash^{n1}$

$\dashv^{1n} ::= ((thresholdBreaker_{1 \lhd n})$

$\pentagon^{n}_{F} ::= (\blacktriangle_{F})^{n}$

$(\blacktriangle_{F})^{n} ::= [\blacktriangle_{F}]_{n}$

$\blacktriangle_{F} ::= \triangle_{F}$

$\triangle_{F} ::=$ «recursiveFib»

$\vdash^{n1} ::= (\oplus_{n \rhd 1})$

D.5 Stencil

The stencil application has been designed as follow:

$\overleftarrow{((customscatter_{1 \lhd n}) \,.\, [\text{«T»}]_{n} \,.\, (swap \& \oplus_{n \rhd 1}))}_{iteration}$

Using the building block grammar represented in listing 3.1, the stencil application is generated as follows:

$\square_{S(T)} ::= \blacktriangle_{S}$

$\blacktriangle_{S} ::= \triangle_{S}$

$\triangle_{S} ::= \overleftarrow{(\dashv^{1n} \,.\, \pentagon^{n}_{S} \vdash^{n1})}_{iteration}$

$\dashv^{1n} ::= (customscatter_{1 \lhd n})$

$\pentagon^{n}_{S} ::= (\blacktriangle_{S})^{n}$

$(\blacktriangle_{S})^{n} ::= [\blacktriangle_{S}]_{n}$

$\blacktriangle_{S} ::= \triangle_{S}$

$\triangle_{S} ::=$ «T»

$\vdash^{n1} ::= (swap \& \oplus_{n \rhd 1})$

D.6 N-queen

The *N-queen* application has been designed as follow:

$(CustomScatter_{1 \lhd n}) \,.\, [\text{«SommersNqueen»}]_{n}$

Using the building block grammar represented in listing 3.1, the stencil application is generated as follows:

$\square_{Nq} ::= \blacklozenge_{Nq}$

$\blacklozenge_{Nq} ::= \diamondsuit_{Nq}$

$\diamondsuit_{Nq} ::= \dashv^{1n} \,.\, \bigcirc^{n}_{Nq}$

$\dashv^{1n} ::= (CustomScatter_{1 \lhd n})$

$\bigcirc^{n}_{Nq} ::= \pentagon^{n}_{Nq}$

$\pentagon^{n}_{Nq} ::= (\blacktriangle_{Nq})^{n}$

$(\blacktriangle_{Nq})^{n} ::= [\blacktriangle_{Nq}]_{n}$

$\blacktriangle_{Nq} ::= \triangle_{Nq}$

$\triangle_{Nq}$ ::= «SommersNqueen»

D.7 EISPACK Routine

The EISPACK application has been designed as follow:

‹|Generation|› .‹|Solution|› .‹|Verification|›

Using the building block grammar represented in listing 3.1, the EISPACK application can be demonstrated as:

$\square_{\text{EISPACK}}$ ::= $\blacktriangle_{GSV}$

$\blacktriangle_{GSV}$ ::= $\triangledown_{GSV}$

$\triangledown_{GSV}$ ::= $\triangle_G$. $\triangledown_{SV}$

$\triangledown_{SV}$ ::= $\triangle_S$. $\triangle_V$

$\triangle_G$ ::= ‹|Generation|›

$\triangle_S$ ::= ‹|Solution|›

$\triangle_V$ ::= ‹|Verification|›

D.8 `getSolution` component for *SMTWTP*

The getSolution component embedded in the *SMTWTP* application has been designed as follow:

$$\overleftarrow{(unicast_{1\lhd 2} \,.\, [scatter_{1\lhd n} \,.\, [\text{«}findSol_{CPU}\text{»}]_n \,.\, gatherall_{n\rhd 1}\ ,}$$
$$\overrightarrow{unicast_{1\lhd m} \,.\, [\text{‹|}\ findSol_{GPU}\ \text{|›}]_m \,.\, gather_{m\rhd 1}] \,.\, (gAll\&g_{best}\&u_{pheromone_{2\rhd 1}}))_{iter}}$$

Using the building block grammar represented in listing 3.1, the getSolution component is generated as follows: $\square_{GetSol}$ ::= $\blacktriangle_{CG}$

$\blacktriangle_{CG}$ ::= $\overleftarrow{\triangle_{CG}}$

$\triangle_{CG}$ ::= $(\dashv^{12} \,.\, \pentagon^n_{CG} \,.\, \vdash^{21})_{iter}$

$\dashv^{12}$::= $unicast_{1\lhd 2}$

$\pentagon^2_{CG}$ ::= $(\blacktriangle_{CG})^2$

$(\blacktriangle_{CG})^2$ ::= [$\blacktriangle_C$ ',' $\blacktriangle_G$]

$\blacktriangle_C$ ::= $\triangle_C$

$\triangle_C$ ::=$\vdash^{1n} \,.\, \pentagon^n_C \,.\, \dashv^{n1}$

$\dashv^{1n}$::= $scatter_{1\lhd n}$

$\pentagon^n_C$ ::= $(\blacktriangle_{Cs})^n$

$(\blacktriangle_{Cs})^n$ ::= $[\blacktriangle_{Cs}]_n$

$\blacktriangle_{Cs}$ ::= $\triangle_{Cs}$

$\triangle_{Cs}$ ::= «$findSol_{CPU}$»

$\vdash^{n1}$::= $gatherall_{n\rhd 1}$

$\blacktriangle_G ::= \triangle_G$

$\triangle_G ::= \dashv^{1m} . \pentagon^m_G . \vdash^{m1}$

$\dashv^{1m} ::= unicast_{1 \lhd m}$

$\pentagon^m_G ::= (\blacktriangle_{Gs})^m$

$(\blacktriangle_{Gs})^m ::= [\blacktriangle_{Gs}]_m$

$\blacktriangle_{Gs} ::= \triangle_{Gs}$

$\triangle_{Gs} ::= \langle \mid findSol_{GPU} \mid \rangle$

$\vdash^{m1} ::= gather_{m \rhd 1}$

D.9 *MD*

The *MD* application has been designed as follow:

$unicast_{1 \lhd 2} . [scatter_{1 \lhd n} . [\langle\langle md_{cpu} \rangle\rangle]_n . gatherall_{n \rhd 1} , unicast_{1 \lhd m} . [\langle \mid md_{gpu} \mid \rangle]_m]$

Using the building block grammar represented in listing 3.1, the *MD* application is generated as Follows:

$\square_{CMD} ::= \blacklozenge_{CG}$

$\blacklozenge_{CG} ::= \diamondsuit_{CG}$

$\diamondsuit_{CG} ::= \dashv^{12} . \bigcirc^n_{CG}$

$\dashv^{12} ::= unicast_{1 \lhd 2}$

$\bigcirc^2_{CG} ::= (\square_{CG})^2$

$(\square_{CG})^2 ::= [\square_C \text{ ',' } \square_G]$

$\square_C ::= \blacktriangle_C$

$\blacktriangle_C ::= \triangle_C$

$\triangle_C ::= \vdash^{1n} . \pentagon^n_C . \dashv^{n1}$

$\dashv^{1n} ::= scatter_{1 \lhd n}$

$\pentagon^n_C ::= (\blacktriangle_{Cs})^n$

$(\blacktriangle_{Cs})^n ::= [\blacktriangle_{Cs}]_n$

$\blacktriangle_{Cs} ::= \triangle_{Cs}$

$\triangle_{Cs} ::= \langle\langle md_{cpu} \rangle\rangle$

$\vdash^{n1} ::= gatherall_{n \rhd 1}$

$\square_G ::= \blacklozenge_G$

$\blacklozenge_G ::= \diamondsuit_G$

$\diamondsuit_G ::= \dashv^{1m} . \pentagon^m_G$

$\dashv^{1m} ::= unicast_{1 \lhd m}$

$\pentagon^m_G ::= (\blacktriangle_{Gs})^m$

$(\blacktriangle_{Gs})^m ::= [\blacktriangle_{Gs}]_m$

$\blacktriangle_{Gs} ::= \triangle_{Gs}$

$$\triangle_{Gs} ::= \langle \mid md_{gpu} \mid \rangle$$

Appendix E

Implementation of N-body Simulation under Three Frameworks

In this section we explain the N-body Simulation algorithm implemented in FastFlow, Thrust and SKePU.

E.1 N-body Simulation

We use the gravitational potential to illustrate the basic form of computation in an all-pairs N-body simulation.

Given N bodies with an initial position x_i and velocity v_i for $1 < i < N$, the total force F_i on body i, due to its interactions with the other $N-1$ bodies, is obtained by summing all interactions f_{ij} on body i caused by its gravitational attraction to body j as expressed in equation (E.1).

$$F_i = \sum_{j=0}^{n} \frac{m_i \times r_{ij}}{(\|r_{ij}\|^2 + \varepsilon^2)^{3/2}} \tag{E.1}$$

Note that m_i and m_j are the masses of bodies i and j respectively; $r_{ij} = x_j - x_i$ is the vector from body i to body j; G is the gravitational constant; and ε is a softening factor to preclude collisions between bodies.

An integrator is used to update the positions and velocities and included in computing the runtime. However, full details of its implementation is omitted since it has a complexity of $O(N)$ and its cost becomes insignificant as N grows.

The energy is the combination of kinetic energy between each pairs of bodies and the potential energy when the bodies are far apart as expressed in equation (E.2).

$$e = \sum_{i=0}^{n} m_i \times \frac{\|v_i\|^2}{2} - \frac{G}{2} \sum_{i=0}^{n} m_i \sum_{j=0 j \neq i}^{n} \frac{m_j}{\|r_i j\|} \tag{E.2}$$

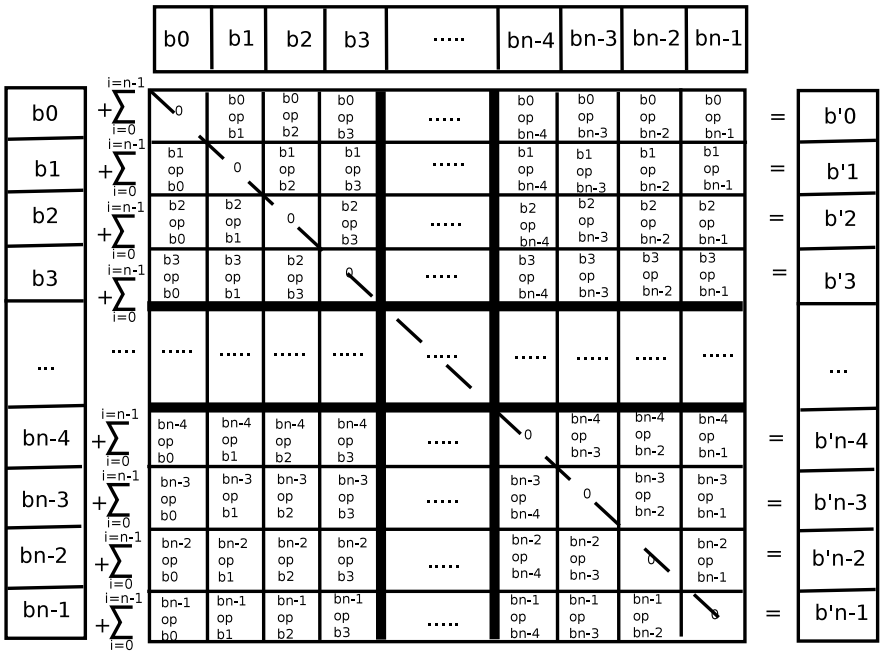


Figure E.1: the visual representation of the intermediate grid for calculating the gravitational computation of the N-body algorithm [126]

A similar technique as that of used for force is applied here. In order to avoid repetition, we only explain the technique used to implement the gravitational force among all-pairs bodies. Each body has 7 properties including the 3 dimensions of position, the 3 dimensions of speed and the weight of the body.

The gravitational potential F has been used to illustrate the basic form of computation in an all-pairs N-body simulation. The gravitational computation of the algorithm can be estimated by calculating each entry f_{ij} in an $N \times N$ grid of all pair-wise forces, called F. Then the total force F_i on body i is obtained from the sum of all entries in row i. Figure E.1 shows this computation [126].

We use the notion of a computational tile, a rectangular region of the grid of pair-wise forces consisting of `HSIZE` rows and `WSIZE` columns as introduced in [126]. Only $\texttt{HSIZE} + \texttt{WSIZE}$ body descriptions are required to evaluate all $\texttt{HSIZE} \times \texttt{WSIZE}$ interactions in the tile (`WSIZE` of which can be reused later). The maximum block reuse is achieved for square tiling where the $\texttt{HSIZE} = \texttt{WSIZE}$.

Once the two bodies b_i and b_j are fetched into the processor registers, the forces of b_i on b_j and b_j on b_i can be calculated at the same time. This reduces the number of memory accesses by half from N^2 times to $\frac{(N \times (N+1))}{2}$ times and, consequently, improves runtime performance of the algorithm. Therefore, we only need to access memory to calculate the upper triangle of the grid in order to compute all of the forces on all bodies [126].

In this case, the grid F is divided into square regions of pair-wise forces consisting of HSIZE rows and HSIZE columns. Each block contains HSIZE threads. In each iteration, each thread t_i from block p_i loads body b_i into the shared memory, *sh*. All threads of the block will use the *sh* to calculate the intermediate results of $\texttt{HSIZE}^2$ cells. The access to global memory for each block is coalesced. This reduces the memory latency and increases the performance considerably. To achieve optimal reuse of data, the computation of a tile is arranged so that the interactions in each row are evaluated in sequential order, updating the acceleration vector, while separate rows are evaluated in parallel. Each thread reads only 2 bodies to calculate the tile of $\texttt{HSIZE}^2$ and therefore $\frac{N}{\texttt{HSIZE}}$ bodies to calculate the force of all bodies on b_i. The total global memory access is equal to $2 \times \texttt{HSIZE} \times (\frac{N}{\texttt{HSIZE}})^2$.

In the following, we explain the optimisation techniques used for each framework.

FastFlow In this instance, a farm pattern is used to provide the gravitational force. The farm is basically composed of an emitter, a number of workers, and a collector. The emitter is used as a load balancer filter which splits F into square blocks of $\texttt{HSIZE} \times \texttt{HSIZE}$ and offloads them to workers queue asynchronously by using the round-robin technique.

To achieve optimal reuse of data, each worker concurrently calculates the force of received square tiles, so that the interactions in each tile are evaluated in sequential order by using the upper triangular technique, generating the partial updated vector. The collector is a filter for farm gatherer and collects the partial updated vector from workers and updates the bodies.

Thrust In Thrust there is no support for complex vector type. Therefore, 7 different vectors should be considered for the properties of bodies. However, there is a patten called *make_tuple* which combines all variables together, similar to an object in C++. This pattern is used to send all properties of bodies as a single object B to the force functor.

B is therefore divided into k blocks of size of $\texttt{HSIZE} = \frac{N}{k}$, by the Thrust distribution policy, where k is the maximum number of available threads in the system.

To implement the gravitational force, the transform pattern is used which maps the functor over each block. As there is no support for 2D operations in Thrust, the tiling approach for calculating F requires that, for each iteration, the same WSIZE block of the B vector is given to each thread as a read-only constant data. Each of these k threads, in parallel, evaluate the force of the assigned $\texttt{HSIZE} \times \texttt{WSIZE}$ tile in sequential order and then update the velocity of B.

It noted that it is not possible to have the triangular approach as the block of WSIZE is a constant read-only block. The tile assignment policy to each thread is synchronous. As HSIZE of the tiles are defined by the Thrust distribution policy, it is not always feasible to have square tilings. This is because HSIZE can be large, and having the same WSIZE can create an extremely-large square tile.

For the GPU backend, each thread is assigned to a body and the number of accesses to memory is equal to N^2. The first N accesses are from global memory, while the next N accesses are from constant memory.

SKePU Similar to Thrust, SKePU does not support the complex type for vectors. Hence, 7 vectors are needed to store the bodies properties. A macro function can accept at most 3 input parameters and 1 constant value. Moreover, unlike Thrust, there is no pattern for combining the bodies properties together. Hence, there is no way to calculate the gravitational force using only one macro since each body has 7 parameters.

Furthermore, unlike Thrust, it is not possible to send a block of data as a constant parameter to a macro. A macro only accepts scalar values as input. So, to calculate F, each entry must be computed independently. In this case we need a minimum of 3 macros and 7 function calls to calculate the gravitational force of b_i on b_j.

For the GPU backend, for each body a thread will be created hence, there are N^2 available parallel processes. However, to calculate each entry of the grid we need to read 2 bodies and the number of access to memory is equal to $14 \times N^2$. This is highly inefficient as it creates maximum redundancy of memory access.

Therefore, in terms of the GPU backend, the main difference between Thrust and SKePU is that in Thrust, N memory accesses are from the constant memory in the GPU. Although the constant memory is slow, it caches the read block and is optimised for broadcast. However, in SKePU, all $14 \times N^2$ accesses are from global memory which is also slow and requires sequential and aligned $16 - byte$ reads and writes to be fast (coalesced read/write). Unfortunately, it is not feasible to have the coalesced access in SKePU because of the aforementioned limitations. Ergo, the inefficient memory access here will be more obvious than in multi-core CPU backend, as memory access in CPU is optimised and automatically caches a block of data in each memory access.

Bibliography

[1] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

[2] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[3] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Gpu-based video feature tracking and matching. In *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, volume 278, page 4321, 2006.

[4] Peter J Lu, Hidekazu Oki, Catherine A Frey, Gregory E Chamitoff, Leroy Chiao, Edward M Fincke, C Michael Foale, Sandra H Magnus, William S McArthur Jr, Daniel M Tani, et al. Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 5(3):179–193, 2010.

[5] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.

[6] Stéphane Bihan, Georges-Emmanuel Moulard, Romain Dolbeau, Henri Calandra, and Rached Abdelkhalek. Directive-based heterogeneous programming–a gpu-accelerated rtm use case. In *Proceedings of the 7th international conference on computing, communications and control technologies*, 2009.

[7] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[8] Johan Enmyren and Christoph W Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[9] Wen-mei W Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., 2011.

[10] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl-a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE, 2011.

[11] Michael D McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 5–5. USENIX Association, 2010.

[12] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[13] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[14] Marco Danelutto and Massimo Torquati. A risc building block set for structured parallel programming. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 46–50. IEEE, 2013.

[15] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming*, pages 1–20, 2013.

[16] Tong Li, Dan Baumberger, David A Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 53. ACM, 2007.

[17] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[18] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[19] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 19(1), pages 120–132. ACM, 1991.

[20] James Lyle Peterson and Abraham Silberschatz. *Operating system concepts*, volume 2. Addison-Wesley Reading, MA, 1985.

[21] Perry H Wang, Jamison D Collins, Gautham N Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *ACM SIGPLAN Notices*, volume 42(6), pages 156–166. ACM, 2007.

[22] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[23] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the international conference on Supercomputing*, pages 120–129. ACM, 2011.

[24] Uri Verner, Assaf Schuster, Mark Silberstein, and Avi Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 8. ACM, 2012.

[25] Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.

[26] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 84–93. ACM, 2012.

[27] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[28] Víctor J Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers*, pages 19–33. Springer, 2009.

[29] Eduard Ayguadé, Rosa M Badia, Francisco D Igual, Jesús Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. In *Euro-Par 2009 Parallel Processing*, pages 851–862. Springer, 2009.

[30] George Teodoro, Rafael Sachetto, Olcay Sertel, Metin N Gurcan, W Meira, Umit Catalyurek, and Renato Ferreira. Coordinating the use of gpu and cpu for improving

performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[31] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 200–209. IEEE, 2009.

[32] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[33] Marco Danelutto and Giorgio Zoppi. Behavioural skeletons meeting services. In *Computational Science–ICCS 2008*, pages 146–153. Springer, 2008.

[34] Marco Aldinucci, Marco Danelutto, Giorgio Zoppi, and Peter Kilpatrick. Advances in autonomic components & services. In *From Grids to Service and Pervasive Computing*, pages 3–17. Springer, 2008.

[35] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Management in distributed systems: a semi-formal approach. In *Euro-Par 2007 Parallel Processing*, pages 651–661. Springer, 2007.

[36] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Towards hierarchical management of autonomic components: a case study. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 3–10. IEEE, 2009.

[37] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Vamis Xhagjika. Libero: a lightweight behavioural skeleton framework. Technical report, Technical Report TR-10-07, Dept. Computer Science, Univ. of Pisa, Italy, 2010.

[38] Manish Agarwal, Viraj Bhat, Hua Liu, Vincent Matossian, V Putty, Cristina Schmidt, Guangsen Zhang, L Zhen, Manish Parashar, Bithika Khargharia, et al. Automate: Enabling autonomic applications on the grid. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 48–57. IEEE, 2003.

[39] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Software Composition*, pages 82–97. Springer, 2006.

[40] Paraphrase: Parallel patterns for adaptive heterogeneous multicore systems. Available: http://goo.gl/DHzGL9.

[41] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, 2011.

[42] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[43] Robert N Bateson. *Introduction to control system technology*. Prentice Hall PTR, 1989.

[44] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[45] Aaftab Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[46] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[47] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[48] Lee Howes, Anton Lokhmotov, Alastair F Donaldson, and Paul HJ Kelly. Towards metaprogramming for parallel systems on a chip. In *Euro-Par 2009–Parallel Processing Workshops*, pages 36–45. Springer, 2010.

[49] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[50] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In *Formal Methods for Components and Objects*, pages 218–236. Springer, 2013.

[51] Horacio González-Vélez. An adaptive skeletal task farm for grids. In *Euro-Par 2005 Parallel Processing*, pages 401–410. Springer, 2005.

[52] Berna L Massingill, Timothy G Mattson, and Beverly A Sanders. Parallel programming with a pattern language*. *International Journal on Software Tools for Technology Transfer*, 3(2):217–234, 2001.

[53] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[54] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[55] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[56] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[57] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[58] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. Fastflow: Efficient parallel streaming applications on multi-core. *arXiv preprint arXiv:0909.1187*, 2009.

[59] Gregory F Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200. ACM, 2008.

[60] Eduard Ayguade, Rosa M Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 154–167. Springer, 2009.

[61] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[62] Yuan Wen, Zheng Wang, and Michael O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *High Performance Computing (HiPC)*, 2014.

[63] Jeffrey Brown, Al Geist, C Pancake, and D Rover. Software tools for developing parallel applications. part 1: Code development and debugging. Technical report, Oak Ridge National Lab., Computer Science and Mathematics Div., TN (United States), 1997.

[64] Jeffrey Brown, Al Geist, C Pancake, and D Rover. Software tools for developing parallel applications. part 2: Interactive control and performance tuning. Technical report,

Oak Ridge National Lab., Computer Science and Mathematics Div., TN (United States), 1997.

[65] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *EuroPar 2009*, volume 5704 of *LNCS*, pages 9–20. Springer, 2009.

[66] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing*, pages 136–151. Springer, 2006.

[67] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[68] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[69] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[70] Takahiro Katagiri, Kenji Kise, Hiroaki Honda, and Toshitsugu Yuba. Fiber: A generalized framework for auto-tuning software. In *High Performance Computing*, pages 146–159. Springer, 2003.

[71] Marco Aldinucci, Alessandro Petrocelli, Edoardo Pistoletti, Massimo Torquati, Marco Vanneschi, Luca Veraldi, and Corrado Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *Euro-Par 2005*, volume 3648 of *LNCS*, pages 771–781. Springer, 2005.

[72] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. Enabling self-management of component based distributed applications. In *From Grids to Service and Pervasive Computing*, pages 163–174. Springer, 2008.

[73] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons for component autonomic management on grids. In *Making Grids Work*, pages 3–15. Springer, 2008.

[74] Marco Aldinucci, Sonia Campa, Marco Danelutto, and Marco Vanneschi. Behavioural skeletons in gcm: autonomic management of grid components. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 54–63. IEEE, 2008.

[75] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[76] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Semi-formal models to support program development: Autonomic management within component based parallel and distributed programming. In *Formal Methods for Components and Objects*, pages 204–225. Springer, 2009.

[77] Horacio González-Vélez and Murray Cole. Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. *Concurrency and Computation: Practice and Experience*, 22(15):2073–2094, 2010.

[78] Michael T. Garba and Horacio González-Vélez. Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: A decentralised queue monitoring strategy. *Parallel Processing Letters*, 22(2):1240008, 2012.

[79] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with FastFlow. In *Euro-Par 2011*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, August 2011. Springer.

[80] Mehdi Goli, Michael T Garba, et al. Streaming dynamic coarse-grained cpu/gpu workloads with heterogeneous pipelines in fastflow. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 445–452. IEEE, 2012.

[81] Mehdi Goli and Horacio González-Vélez. N-body computations using skeletal frameworks on multicore cpu/graphics processing unit architectures: an empirical performance evaluation. *Concurrency and Computation: Practice and Experience*, 26(4):972–986, 2014.

[82] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. Paraphrasing: Generating parallel programs using refactoring. In *Formal Methods for Components and Objects*, pages 237–256. Springer, 2013.

[83] Douglas Crockford. Json: The fat-free alternative to xml. In *Proc. of XML*, volume 2006, 2006.

[84] Fastflow contributors. Available: http://goo.gl/2tWvVK.

[85] Mehdi Goli and Horacio González-Vélez. Heterogeneous Algorithmic Skeletons for Fast Flow with Seamless Coordination over Hybrid Architectures. In *PDP*, pages 148–156. IEEE Computer Society, 2013.

[86] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[87] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Euro-Par 2012 Parallel Processing*, pages 662–673. Springer, 2012.

[88] B J Nelson. *Remote procedure call*. PhD thesis, Pittsburgh Univ., Pittsburgh, PA, 1981. Presented on May 1981.

[89] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[90] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. The opencl programming book. *Fixstars Corporation*, 63, 2010.

[91] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[92] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[93] Mehdi Goli, John McCall, Christopher Brown, Vladimir Janjic, and Kevin Hammond. Mapping parallel programs to heterogeneous cpu/gpu architectures using a monte carlo tree search. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 2932–2939. IEEE, 2013.

[94] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[95] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

[96] Guillaume Chaslot. *Monte-carlo tree search*. PhD thesis, PhD thesis, Maastricht Univ, 2010.

[97] O. Fialka and M. Cadik. FFT and convolution performance in image filtering on GPU. In *10th Int. Conference on Information Visualization*, pages 609–614, London, july 2006. IEEE.

[98] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M LaConte. Medical image processing on the gpu–past, present and future. *Medical image analysis*, 17(8):1073–1094, 2013.

[99] OR Vincent and O Folorunso. A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science & IT Education Conference (InSITE)*, pages 97–107, 2009.

[100] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846. IEEE, 1998.

[101] Ioannis Pitas. *Digital image processing algorithms and applications*. John Wiley & Sons, 2000.

[102] Rachid Deriche. Recursively implementating the gaussian and its derivatives. *Proc. 2nd International Conference on Image Processing, Singapore*, pages 263–267, 1992.

[103] Hans-Wolfgang Loidl and Jeremy Singer. Sicsa multicore challenge editorial preface. *Concurrency and Computation: Practice and Experience*, 26(4):929–934, 2014.

[104] Bodil Branner. The mandelbrot set. In *Proc. symp. appl. math*, volume 39, pages 75–105, 1989.

[105] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[106] Verner E Hoggatt. *Fibonacci and Lucas numbers*. Houghton Mifflin Boston, 1969.

[107] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[108] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[109] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[110] Jeff Somers. The n queens problem-a study in optimization, 2002.

[111] Michael Garba, Horacio González-Vélez, and Daniel Roach. GPU Acceleration for Hermitian Eigensystems. *Transactions on Computational Collective Intelligence*, 2012. In Press.

[112] Christopher Brown, Vladimir Janjic, Kevin Hammond, Kamran Idrees, Colin Glass, Amer Wafai, Mehdi Goli, and John McCall. Bridgind the divide a new methodology for semi-automatic programming of heterogeneous parallel machines. In *CSRD springer, submitted*, 2014.

[113] Matthijs Den Besten, Thomas Stützle, and Marco Dorigo. An ant colony optimization application to the single machine total weighted tardiness problem. In *Proceedings of ANTS*, pages 39–42, 2000.

[114] Michael P Allen. Introduction to molecular dynamics simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins*, 23:1–28, 2004.

[115] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *Euro-Par 2012: Parallel Processing Workshops*, pages 47–56. Springer, 2013.

[116] Sonia Campa, Marco Danelutto, Mehdi Goli, Horacio González-Vélez, Alina Madalina Popescu, and Massimo Torquati. Parallel patterns for heterogeneous cpu/gpu architectures: Structured parallelism from cluster to cloud. *Future Generation Computer Systems*, 37:354–366, 2014.

[117] Suresh Boob, Horacio Gonzalez-Velez, and Alina Madalina Popescu. Automated instantiation of heterogeneous fast flow cpu/gpu parallel pattern applications in clouds. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 162–169. IEEE, 2014.

[118] Tudor Serban, Marco Danelutto, and Peter Kilpatrick. Autonomic scheduling of tasks from data parallel patterns to cpu/gpu core mixes. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 72–79. IEEE, 2013.

[119] Aldinucci M, Torquati M, Drocco M, Peretti Pezzi G, and Spampinato C. Fastflow: Combining pattern-level abstraction and efficiency in gpgpus. *in GPU Technology Conference,GTC*, 2014.

[120] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

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

[122] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[123] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, and Guillermo Ordonez. Mallba: a software library to design efficient optimisation algorithms. *International Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.

[124] Mariusz Nowostawski and Riccardo Poli. Parallel genetic algorithm taxonomy. In *Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference*, pages 88–92. IEEE, 1999.

[125] Kevin Hammond, Abdallah Al Zain, Gene Cooperman, Dana Petcu, and Phil Trinder. Symgrid: a framework for symbolic computation on the grid. In *Euro-Par 2007 Parallel Processing*, pages 457–466. Springer, 2007.

[126] L. Nyland, M.Harris, and J.Prins. Fast N-body simulation with CUDA. *GPU Gems 3. H. Nguyen*, 2007.