UNIVERSITY OF HERTFORDSHIRE

An Efficient Execution Model for Reactive Stream Programs

by

Vu Thien Nga Nguyen

A thesis submitted to the University of Hertfordshire
in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

25th September 2014

Abstract

Stream programming is a paradigm where a program is structured by a set of computational nodes connected by streams. Focusing on data moving between computational nodes via streams, this programming model fits well for applications that process long sequences of data. We call such applications *reactive stream programs* (RSPs) to distinguish them from stream programs with rather small and finite input data.

In stream programming, concurrency is expressed implicitly via communication streams. This helps to reduce the complexity of parallel programming. For this reason, stream programming has gained popularity as a programming model for parallel platforms. However, it is also challenging to analyse and improve the performance without an understanding of the program's internal behaviour. This thesis targets an efficient execution model for deploying RSPs on parallel platforms. This execution model includes a monitoring framework to understand the internal behaviour of RSPs, scheduling strategies for RSPs on uniform shared-memory platforms; and mapping techniques for deploying RSPs on heterogeneous distributed platforms. The foundation of the execution model is based on a study of the performance of RSPs in terms of throughput and latency. This study includes quantitative formulae for throughput and latency; and the identification of factors that influence these performance metrics.

Based on the study of RSP performance, this thesis exploits characteristics of RSPs to derive effective scheduling strategies on uniform shared-memory platforms. Aiming to optimise both throughput and latency, these scheduling strategies are implemented in two heuristic-based schedulers. Both of them are designed to be centralised to provide load balancing for RSPs with dynamic behaviour as well as dynamic structures. The first one uses the notion of positive and negative data demands on each stream to determine the scheduling priorities. This scheduler is independent from the runtime system. The second one requires the runtime system to provide the position information for each computational node in the RSP; and uses that to decide the scheduling priorities. Our experiments show that both schedulers provides similar performance while being significantly better than a reference implementation without dynamic load balancing.

Also based on the study of RSP performance, we present in this thesis two new heuristic partitioning algorithms which are used to map RSPs onto heterogeneous distributed platforms. These are Kernighan-Lin Adaptation (KLA) and Congestion Avoidance (CA), where the main objective is to optimise the throughput. This is a multi-parameter optimisation problem where existing graph partitioning algorithms are not applicable. Compared to the generic meta-heuristic *Simulated Annealing* algorithm, both proposed algorithms achieve equally good or better results. KLA is faster for small benchmarks while slower for large ones. In contrast, CA is always orders of magnitudes faster even for very large benchmarks.

Acknowledgements

First of all, I would like to express my great gratitude to my father, who unfortunately can not see this day. He once said to me "Con là hoài bão của ba" (Your success is my dream) and I have carried that saying with me during my master and PhD studies. For my weakest moments of being homesick, being scared in foreign countries, and feeling despaired when facing technical problems, his encouragement has helped me to go over them not pleasantly but strongly to reach this day.

I would like to express my deepest gratitude to my supervisors, Dr. Raimund Kirner and Prof. Alex Sharafanko, for their time and support. I am very grateful for Dr. Raimund Kirner for valuable and enjoyable discussions. His patience and insightful comments on my research ideas had an important impact on this thesis. I have learnt from him not only precious knowledge but also several useful skills, especially technical writing and communication.

I would like to thank all my colleagues in the Compiler Technology and Computer Architecture group, for all their technical support as well as many parties and fun times together. I would particularly thank Frank Penczek and Michael Zolda for long technical discussions and useful tips in writing and presentation.

Furthermore, I would like to thank all my family for their support and all my friends for great time with lots of laugh. Finally, lots and lots of thanks to my husband, Daniel Rolls, for his understanding, tolerance, encouragement and especially for proofreading this thesis.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii

Abbreviations xv

1 Introduction 1
- 1.1 Reactive Stream Programs . . . 2
- 1.2 Execution Model for Reactive Stream Programs . . . 2
- 1.3 Evaluation of Reactive Stream Programs . . . 3
- 1.4 Research Questions . . . 5
- 1.5 Contributions . . . 5
  - 1.5.1 Publications . . . 6
- 1.6 Structure of the Thesis . . . 7
- 1.7 Chapter Summary . . . 8

2 Background 11
- 2.1 Data Flow Programming . . . 11
- 2.2 Stream Programming . . . 12
  - 2.2.1 Stream . . . 12
  - 2.2.2 Stream-Programming Model . . . 12
  - 2.2.3 Properties of Stream Programs . . . 14
- 2.3 Interactive/Reactive Systems . . . 15
- 2.4 Context of this Thesis: Reactive Stream Programs . . . 16
  - 2.4.1 Data in Reactive Stream Programs . . . 18
  - 2.4.2 Execution Model of RSPs . . . 18
  - 2.4.3 Message Derivation . . . 20
  - 2.4.4 Message Completion . . . 21
- 2.5 Instantiation of the Execution Model for Reactive Stream Programs . . . 22
  - 2.5.1 Stream programming with S-Net . . . 22
  - 2.5.2 S-Net Compiler and Runtime System . . . 24

2.5.3 LPEL — A User-mode Microkernel for the Streaming Language S-Net . . . 26
2.5.4 S-Net on Distributed Systems . . . 27
2.6 Chapter Summary . . . 30

3 Related Work 33
3.1 Monitoring Parallel Programs . . . 33
3.2 Taxonomy of Scheduling Methods . . . 35
3.2.1 Offline Scheduling . . . 35
3.2.2 Online Scheduling . . . 36
3.2.2.1 Centralised Scheduling . . . 36
3.2.2.2 Distributed Scheduling . . . 36
3.2.2.3 Centralised Mediation . . . 37
3.3 Scheduling Reactive Stream Programs on Many-Core Systems . . . 38
3.3.1 Static Scheduling . . . 38
3.3.2 Dynamic Scheduling . . . 41
3.4 Scheduling Stream Programs on Distributed Systems . . . 43
3.5 Graph Partitioning . . . 45
3.6 Chapter Summary . . . 47

4 Use Cases 49
4.1 Data Encryption Standard - DES . . . 49
4.2 Ant Colony Optimization - ANT . . . 50
4.3 Ray Tracing - RT . . . 52
4.4 Fast Fourier Transform - FFT . . . 52
4.5 Color Histogram Calculation - HIST . . . 54
4.6 Image Filtering - IMF . . . 55
4.7 Object Detecting - OBD . . . 56
4.8 Moving Target Indicator - MTI . . . 57
4.9 Monte Carlo Option Price - MC . . . 58

5 Performance Analysis for Reactive Stream Programs 60
5.1 Performance Metrics . . . 60
5.1.1 Throughput . . . 60
5.1.2 Latency . . . 61
5.2 Performance with Different Arrival Rate . . . 62
5.2.1 Theoretical Analysis . . . 62
5.2.2 Experimental Verification . . . 63
5.3 Quantitative Analysis of Performance . . . 67
5.3.1 Throughput Analysis . . . 68
5.3.1.1 Uniformed Shared Memory Platforms . . . 68
5.3.1.2 Distributed Platforms . . . 69
5.3.2 Latency Analysis . . . 71
5.4 Chapter Summary . . . 72

6 Monitoring of Reactive Stream Programs 73
6.1 Conceptions of the Monitoring Framework . . . 73
6.1.1 Monitoring the Runtime System . . . 73

6.1.2 Monitoring the Execution Layer . . . 74
6.2 Potential Benefits of the Monitoring Framework . . . 75
6.2.1 Performance Metric Measurement . . . 76
6.2.2 Extracting RSP Properties . . . 77
6.2.3 Automatic Load Balancing . . . 79
6.2.4 Bottleneck Detection . . . 80
6.3 Implementation of the Monitoring Framework in S-Net and LPEL . . . 81
6.3.1 Instrumenting the S-Net runtime system . . . 81
6.3.2 Instrumenting the LPEL Execution Layer . . . 82
6.3.3 Operation Modes . . . 84
6.4 Evaluation of the Monitoring Framework . . . 84
6.5 Chapter Summary . . . 87

7 Exploiting the Properties of RSPs for Efficiently Scheduling on Uniform Shared Memory Platforms 89
7.1 Guidelines For Scheduler Design . . . 89
7.1.1 Throughput Optimisation . . . 90
7.1.2 Latency Optimisation . . . 90
7.2 Heuristic Scheduling Strategies for Performance Optimisation on Symmetric Processors . . . 90
7.2.1 Space scheduler . . . 91
7.2.2 Time Scheduler . . . 91
7.2.2.1 Position-based Task Priority . . . 92
7.2.2.2 Demand-based Task Priority . . . 93
7.2.2.3 Scheduling Cycle . . . 94
7.2.3 Scheduling Design Comparison . . . 94
7.3 Implementation of the heuristic priority functions . . . 95
7.3.1 Position-based Task Priority Function . . . 97
7.3.2 Demand-based Priority Function . . . 99
7.4 Evaluation: CS-dbp vs CS-pbp . . . 100
7.4.1 Conceptional Comparison . . . 100
7.4.2 Implementation Comparison . . . 101
7.4.3 Experimental Comparison . . . 102
7.4.3.1 Experiment Set Up . . . 102
7.4.3.2 Experiment Result . . . 105
7.5 Evaluation: CS-dbp vs Default LPEL Scheduler . . . 105
7.5.1 Experimental Set Up . . . 105
7.5.2 Performance Comparison . . . 108
7.5.3 Scalability Comparison . . . 111
7.6 Evaluation: CS-dbp vs Centralised Scheduler with Random Priority . . . 111
7.7 Chapter Summary . . . 112

8 Mapping Reactive Stream Programs onto Distributed Systems 115
8.1 Mapping RSPs onto Distributed Platforms by Graph Partitioning . . . 116
8.2 Usage of Graph Partitions to Optimise Throughput of RSPs . . . 117
8.2.1 Problem Statement . . . 117

8.2.2 Partitioning RSPs with Variable Node Computational Behaviour and Dynamic Program Structures . . . 119
8.3 New Graph Partition Algorithms to Optimise Throughput of RSPs . . . 119
8.3.1 KL-Adapted Algorithm . . . 119
8.3.2 Congestion Avoidance Partitioning Algorithm . . . 126
8.3.3 Local Optima in Heuristic Search . . . 127
8.4 Evaluation of the Partitioning Algorithms . . . 128
8.4.1 Experimental Setup . . . 128
8.4.2 Convergence Speed of KLA and CA . . . 130
8.4.3 Comparison with Simulated Annealing . . . 131
8.5 Chapter Summary . . . 135

9 An Efficient Execution Model for Reactive Stream Programs 137
9.1 Execution Model . . . 137
9.1.1 Overview . . . 137
9.1.2 Integration into S-Net and LPEL . . . 139
9.2 Evaluation . . . 140
9.2.1 Experimental Set Up . . . 140
9.2.2 Performance on Distributed Platforms . . . 141
9.2.3 Performance on shared memory platforms . . . 145
9.3 Chapter Summary . . . 147

10 Conclusion and Outlook 149
10.1 Thesis Summary . . . 149
10.2 Outlook . . . 152

Bibliography 155

List of Figures

2.1 Abstract structure of interactive/reactive Systems 16
2.2 Example: program structure of an image filter application 17
2.3 An example of message derivation 21
2.4 S-Net implementation of the image filter application 22
2.5 An overview of the S-Net execution model using LPEL 25
2.6 Runtime expansion of network combinators in S-Net 26
2.7 The runtime component network of the image filter application (with three instances of box *Filter*) 26
2.8 Execution model of distributed S-Net 28
2.9 Example of an S-Net program being deployed on a distributed system . . 29

4.1 S-Net implementation of the DES encryption 50
4.2 S-Net implementation of the ant colony optimisation 51
4.3 S-Net implementation of the ray tracing application 52
4.4 S-Net implementation of the FFT algorithm 53
4.5 S-Net implementation of the application of histogram calculation 54
4.6 S-Net implementation of the image filter application 55
4.7 S-Net implementation of object detector application 56
4.8 S-Net implementation of the moving target indicator application 58
4.9 S-Net implementation of the Monte Carlo option price application 59

5.1 Theoretical variation of latency and throughput in different arrival rate ranges 62
5.2 Throughput within different ranges of arrival rate 64
5.3 Overall latency and queuing latency in the underuse and operational ranges 65
5.4 Overall latency of individual external input messages of DES on CS-dbp 66

6.1 Monitoring framework 74
6.2 Latency of an individual task 76
6.3 Deployment of automatic load balancing 80
6.4 The monitoring framework implementation in S-Net and LPEL 81
6.5 Node executions within a task execution 83
6.6 The overhead of the monitoring framework in time and space 86

7.1 Examples of RSP with multiple entry tasks and multiple exit tasks 92
7.2 Design of heuristic stream schedulers for performance optimisation 96
7.3 Runtime components with location vector of $A \star \{\langle stop \rangle\}$ 98
7.4 Runtime components with location vectors of the image filter application 98

7.5 Throughput convergence of IMF on CS-dpb when increasing the number of external input messages . . . 102
7.6 Normalised throughput and latency of CS-dbp and CS-pbp on various applications . . . 106
7.7 Normalised peak throughput and processing latency (with $\lambda = TP_{peak}$) of CS-dbp and DS on various applications . . . 109
7.8 Processing latency comparison with different arrival rates . . . 110
7.9 Performance of HIST using CS-dbp and CS-drp . . . 112

8.1 Graph partitioning and mapping in two phases . . . 116
8.2 Graph partitioning with integrated mapping . . . 117
8.3 Flowchart of the KL-Adapted partitioning algorithm . . . 121
8.4 Flowchart of the Congestion Avoidance partitioning algorithm . . . 124
8.5 Comparison of KLA and CA in the average number of passes . . . 130
8.6 Execution time ratio: et_{KLA}/et_{CA} . . . 131
8.7 Execution time ratio of 50-run CA and KLA to SA ($et_{50\times CA}/et_{SA}$; $et_{50\times KLA}/et_{SA}$) . . . 133
8.8 Achieved quality throughput estimates of 50-run CA and KLA compared to SA . . . 133
8.9 Convergence speed of SA, CA and KLA on MTI_{16} over time . . . 134

9.1 Overview of the new execution model . . . 138
9.2 New execution model of S-Net using LPEL . . . 139
9.3 Main S-Net structure of the MTI application . . . 140
9.4 Throughput of the MTI application: actual value vs estimation of the CA partitioner . . . 142
9.5 Throughput of the MTI application when LPEL conductor and workers run on exclusive CPU cores . . . 143
9.6 Difference in task weight of MTI on 1-machine and 16-machine targets . . . 144
9.7 Average latency of MTI on distributed platforms when LPEL conductors and workers run on exclusive CPU cores . . . 145
9.8 Performance of MTI on shared memory platform . . . 146

List of Tables

2.1 Properties of stream programs 14
2.2 Properties of RSPs supported in the dissertation 17

6.1 Monitoring information needed by different monitoring use cases 75
6.2 Application properties running on a 48-core machine 85

7.1 Values of a location vector element 97
7.2 Priority functions of the middle tasks 100
7.3 Set up for experiment CS-dbp vs CS-pbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-dbp 103
7.4 Set up for experiment CS-dbp vs CS-pbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-pbp 104
7.5 Set up for experiment DS vs CS-dbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in DS 107
7.6 Set up for experiment DS vs CS-dbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-dpb 108

8.1 Semantics of built-in functions used in KLA and CA algorithms 120
8.2 Number of vertices and edges of evaluation benchmarks 129

Abbreviations

CA	**C**ongestion **A**voidance
CRI	**C**ommon **R**untime **I**nterface
CTQ	**C**entral **T**ask **Q**ueue
DF	**D**ata **F**etcher
FIFO	**F**irst **I**n **F**irst **O**ut
ID	**ID**entifier
IM	**I**nput **M**anager
KL	**K**ernighan-**L**in
KPN	**K**ahn **P**rocess **N**etwork
LPEL	**L**ight-weight **P**arallel **E**xecution **L**ayer
MDG	**M**essage **D**erivation **G**raph
MPI	**M**essage **P**assing **I**nterface
OM	**O**utput **M**anager
PE	**P**rocessing **E**lement
RC	**R**untime **C**omponent
RSP	**R**eactive **S**tream **P**rogram
RTS	**R**un**T**ime **S**ystem
SISO	**S**ingle **I**nput **S**ingle **O**utput
SDF	**S**ynchronous **D**ata **F**low
UID	**U**niversal **ID**entifier

Chapter 1

Introduction

Physical barriers in making processors faster by increasing the clock frequency and the need for energy-efficient computing have paved the way for many-core computing to become mainstream. While parallel programming has a long tradition in the field of scientific computing, it has now become an issue of general software development. Stream programming model has become an active research topic, as it provides many practical benefits for parallel programming. For example, it makes some forms of parallelism explicit and the communication over streams facilitates implicit synchronisation.

However, while hiding the intricate issues of parallel programming from the programmer, stream programming makes it complicated to control the behaviour of programs. The performance of stream programs depends highly on the underlying execution model. Therefore, there is a demand for an efficient execution model that can boost the performance of stream programs.

Although there are several research projects in this area, most of them focus on a narrow class of stream programs where their behaviours are predictable. With this thesis, we intend to contribute to the state of the art in a specific aspect of generic stream programs that process virtually infinite sequences of data. We refer to such stream programs as *reactive stream programs* (RSPs). We shall focus our efforts on the study of RSP performance in terms of throughput and latency. This study is the foundation to efficiently schedule RSPs on parallel platforms.

In this introductory chapter, we provide an overview of the context and motivation of our research, and then give an outline of our contributions and the structure of this thesis.

1.1 Reactive Stream Programs

In stream programming, programs are constructed by computational nodes connected by communication channels. Following the data flow model, each computation node can be executed as soon as data is available in its input streams. Streams are communication channels to transfer sequences of data among computation nodes. As a data-centric model, stream programming fits well for applications that process long or infinite sequences of data, for example, video and digital signal processing applications.

Traditionally, a program is transformational in the sense that it accepts an input, performs a transformation, returns an output and terminates. These systems usually do not map well onto stream programs where the inputs are continuously coming from external environments. Stream programs are usually built as reactive systems which interacts continuously with their external environment.

To distinguish from those programs with inputs that are rather small and finite, we define here **Reactive Stream Programs** as stream programs which continuously interact and receive inputs from the environment. As this term is frequently used in the rest of this thesis, it is abbreviated as **RSP** for convenience.

1.2 Execution Model for Reactive Stream Programs

With the current trend towards an increasing number of execution units running in parallel, stream programming has gained more attention for bringing some practical benefits in parallel programming. In particular, it provides explicit forms of parallelism such as pipeline parallelism and data parallelism. It also facilitates implicit synchronisation via stream communication. This helps to relieve the programmer from explicitly managing concurrent communication and synchronisation at the same time. Because of these advantages, several research projects have introduced stream programming frameworks such as StreamIt [TKA02] which follows the principle of Synchronous Data Flow [LM87a], SPADE [GAW+08], and S-Net [GSS08] to name a few.

However, implicit synchronisation has made it difficult to analyse the internal behaviour of programs. With continuous sequences of input data, it becomes even more complicated because of the overlap in processing different input data at the same time. Without the knowledge of the internal behaviour, it is difficult to derive the performance, and even more challenging to build an efficient execution model to obtain the optimal performance for RSPs.

To obtain good performance for RSPs on parallel platforms, the above mentioned projects have introduced execution models with different scheduling strategies. The main trend is to fix the program's behaviour, i.e. to make it predictable. This is the case of the *Synchronous Data Flow* (SDF) model where the structure of an RSP is static and the data rate on each stream is predefined [LM87a]. These properties make it possible to construct the execution of an SDF program as an iteration of its *periodic schedule* [LM87b]. The periodic schedule of an SDF program consists of a number of executions for each computational node in the SDF program. The problem of scheduling an SDF program now becomes the problem of scheduling its periodic schedule. There have been several techniques to derive a static schedule of the periodic schedule on different target platforms. Some examples includes declustering [SL93], heuristic partitioning methods [GTK+02, GTA06], integer linear programming [KM08], machine learning [WO10], approximation [FKBS11], and model checking [MG13].

While these above approaches limit the range of applications to those with static structures and static data rates on stream communications, some other approaches target more general applications by observing the behaviour during runtime. For example, [ZLRA08] and [ME13] are centralised approaches that analyse the runtime behaviour and make dynamic scheduling decisions based on that analysis. A few other approaches like [CC09] combine static analysis based on the profiled behaviour and dynamic adaptation based on the observed behaviour at runtime. There are some other approaches proposed for specific types of stream programs with their own behavioural properties. These include COLA [KHP+09] for SPADE programs, and LPEL [Pro10] for S-Net programs.

1.3 Evaluation of Reactive Stream Programs

Usually, the performance of a program is measured by metrics that directly interest the end user of the program. In RSPs where long sequences of input data arrive continuously, these metrics include throughput and latency. Beside direct metrics, RSPs are often evaluated via indirect performance indicators.

All the approaches mentioned in the previous section aim to achieve the best performance. However, most of them use indirect performance indicators rather than using performance metrics directly. One of the most common indicators is load balancing. Although it is beyond doubt that load balancing is useful for performance in parallel programming, it is non-trivial to understand how it affects the direct performance of RSPs. With a small finite amount of input data, the amount of computation load is fixed. In this case, the load balancing principle, which is to keep all the resources busy all

the time, can guarantee the shortest execution time when the overhead is negligible. In the case of RSPs where the sequence of input data is potentially infinite, that assurance no longer holds. A bad scheduler could keep resources busy by taking in a large amount of input data. Then instead of continuing to process those input data, the scheduler could keep taking new input data. This bad design could either cause the system to fail when the amount of input data exceeds the available memory, or lead to a very long response time.

In an SDF program, an execution of its periodic schedule includes taking in a finite amount of input data and processing all of them before a new execution is carried out for new input data. Since the workload of a periodic schedule of an SDF program is finite, load balancing can be used as a guideline to obtain the minimum execution time of the periodic schedule. As the execution of an SDF program is a repetition of its periodic schedule, the execution time of its periodic schedule is effectively an inversion of the throughput. For this reason, most of the approaches for SDF choose load balancing as their optimising objective, for example [TKM+02] and [KM08]. [FKBS11] is one of the rare approaches to use throughput directly as the optimising objective. Targeting embedded systems, the work in [KTA03] focuses on not only the execution time of the periodic schedule but also on the buffer size and the code size.

Aiming for the Kahn network process [Kah74] with some SDF properties, the approach in [CRA09] uses a cost function combined with the convexity constraint as the guideline for performance. Although it is not clearly stated, the cost function turned out to be the inverse of the throughput. The convexity constraint is claimed to optimise the memory requirement and the latency although there is no proven or clear reasoning.

The approaches [ME13] and [CC09] focus on RSPs with stateless computational nodes, i.e. the node's outputs depend only on their inputs, but not on the history of the node itself. Aiming for load balancing, these approaches use the notion of data demand, which is the amount of data on each stream. The authors of [ZLRA08] also recommend the use of the concept of *data demand* to schedule general RSPs although their concrete method is not clearly presented. The experimental results of these three approaches show a potential usage of data demand. However, these three approaches lack the deep analysis necessary to show the level of impact of data demand on the performance metrics. In addition, there are possibilities for exploring factors other than the data demand that can tune the performance.

1.4 Research Questions

To efficiently deploy RSPs on parallel platforms, it is required to understand the behaviour of the RSPs and its influence on the performance. With implicit synchronisations and the continuous arrival of input data, it is a challenge to devise an efficient execution model for RSPs on parallel platforms.

Initial proposed approaches tend to be limited towards RSPs with static behaviour. This makes it simpler to understand the performance and to develop efficient scheduling methodologies, although it narrows the range of applications. To support a less restricted class of RSPs, later approaches choose to observe the behaviour instead of making them fixed. Yet there is a need for an in-depth study of RSP behaviour and its influence on the performance.

This thesis is motivated by the following research question:

What is an efficient execution model for general reactive stream programs?

This question is fractured into the following sub-questions:

1. Which behavioural factors have influences on the performance of RSPs?
2. How can these behavioural factors be captured?
3. What are the strategies to optimise the performance of RSPs on parallel platforms?

1.5 Contributions

This thesis aims to answer the research question proposed in the previous section via major contributions as follows:

- **Deriving the concepts of throughput and latency in RSPs**. We conduct an in-depth study of the performance in terms of throughput and latency of RSPs. This includes theoretical reasoning and experimental verification of their manners. We also show a deep quantitative analysis of the performance of RSPs on both shared memory and distributed platforms.
- **Capturing the behaviour of RSPs**. Secondly, we identify information required to understand the behaviour of RSPs. This information is essential for the performance calculations, profiling and tuning the scheduler to obtain the optimal

performance. We also present a monitoring framework to capture this information.

- **Exploiting the properties of stream programs to efficiently schedule RSPs on uniform shared-memory platforms**. Thirdly, based on the performance analysis, we derive two novel approaches of using the properties of RSPs to tune performance. The first approach utilises the data demands on stream communications while the second one makes use of the structural position of tasks in the RSP. Since these features are observable at runtime, these two approaches support general RSPs, especially with variable behaviours and dynamic structures. This is a particular challenge for static scheduling based on formal constraints or probabilities.

- **Exploiting graph partitioning algorithms for mapping RSPs onto heterogeneous distributed systems**. Finally, we introduce the use of graph partitioning to map RSPs onto heterogeneous distributed systems. Since existing graph partitioning algorithms are not adequate for this problem domain, we develop two new partitioning algorithms. Employing the performance analysis, these two new algorithms can capture the problem of mapping RSPs to optimise the throughput. These new graph partitioning algorithms are only applicable to RSPs with relatively stable behaviours and relatively static structures. In cases where these features are highly varied during runtime, the propose graph partitions can be used to find the initial mapping. During runtime, an adaptation method is required to repartition the RSP when necessary.

The first contribution answers the first sub-question, i.e. identify the behaviour factors that have influences on the performance of RSPs. The second contribution answers the second sub-question by presenting techniques to capture behavioural factors required to optimise the performance of RSPs. The last two contributions answer the third sub-question, i.e. propose scheduling strategies to optimise the performance of RSPs on both shared memory platforms and distributed platforms.

1.5.1 Publications

Most of the work in this thesis has been published in the following papers:

- Vu Thien Nga Nguyen, Raimund Kirner, Frank Penkzek, *""Monitoring Framework for Stream-processing Networks"*, In HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, Paris, France, 2012.

- Vu Thien Nga Nguyen, Raimund Kirner, Frank Penkzek, *"A Multi-level Monitoring Framework for Stream-based Coordination Programs"*, In Proc. of 12th International Conference on Algorithms and Architectures for Parallel Processing, Fukuoka, Japan, Sep. 2012.
- Vu Thien Nga Nguyen, Raimund Kirner, *"Influences on Throughput and Latency in Stream Programs"* In 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures, Berlin, Germany, 2013.
- Vu Thien Nga Nguyen, Raimund Kirner, *"A Heuristic Strategy for Performance Optimisation of Stream Programs"* In Proc. of IEEE 19th International Conference on Parallel and Distributed Systems, Seoul, Korea, December, 2013.
- Vu Thien Nga Nguyen, Raimund Kirner, *"Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms"*, In Proc. of 13th International Conference on Algorithms and Architectures for Parallel Processing, Sorrento Peninsula, Italy, Dec. 2013.
- Vu Thien Nga Nguyen, Raimund Kirner, *"Throughput Optimisation of Stream Programs on Heterogeneous Distributed Platforms by Graph Partitioning"*, Submitted to IEEE Transactions on Parallel and Distributed Systems, Jun. 2014.

1.6 Structure of the Thesis

The remainder of this thesis is organised as follows.

Chapter 2 provides some background in the context of this thesis — reactive stream programs. This includes data flow programming, stream programming, and reactive systems. This chapter also shows an abstract execution model for RSPs. In addition, it provides key details of an instantiation of the RSP execution model, which is used as the main experimental environment for later chapters.

Chapter 3 discusses related work of this thesis. It reviews scheduling methodologies of RSPs on many-core systems including both static and dynamic techniques. The chapter also describes mapping strategies for RSPs on distributed systems. Also, some approaches that target models similar to RSPs are also included.

Chapter 4 introduces a collection of use cases of RSPs. These use cases will be used as experimental benchmarks for latter chapters.

Chapter 5 investigates the performance of RSPs. Common performance metrics which interest the end user are throughput and latency. The chapter provides the formal

definitions of these metrics and shows how they change in accordance with the arrival rate of input data. In addition, we derive formulas for throughput and latency of RSPs on shared memory and distributed systems.

Chapter 6 discusses behavioural information of an RSP to be captured; and how this information is used in different scenarios such as performance calculation, profiling, bottleneck detection and automatic load balancing. The chapter also presents the conceptional design and the implementation of a monitoring framework to capture this information.

Chapter 7 demonstrates the usage of performance analysis to optimise throughput and latency of RSPs on uniform shared memory platforms. We introduce guidelines for designing efficient schedulers for RSPs on uniform shared memory platforms. The guidelines are implemented in two schedulers. The first scheduler makes use of positive and negative demands on stream communications and is independent from the runtime system. The second scheduler employs the position of computation nodes in the RSP. This approach is based on the awareness of the implementation language and the runtime system.

Chapter 8 introduces the usage of graph partitioning to map RSPs onto heterogeneous distributed systems. It shows how to formulate the graph partitioning problem by using the throughput formula of RSPs. The chapter also develops graph partitioning algorithms for efficiently mapping RSPs onto heterogeneous distributed systems.

Chapter 9 unifies the solution presented in previous chapters and presents a new efficient execution model for RSPs. This execution model integrates the monitoring framework in Chapter 5, partitioning methods in Chapter 7 and the centralised scheduler in Chapter 6.

Chapter 10 finally summarises the work done and discusses the strong points and the weak points of the proposed approaches. It also gives directions for future research in this field.

1.7 Chapter Summary

In this chapter, we have provided an overview of the context and motivation of our research. This thesis focuses on reactive stream programs (RSPs) which are stream programs that process virtually infinite sequences of data. The thesis is motivated by the research question: 'What is an efficient execution model for general reactive stream programs?'. This research question has been fractured into three sub-questions: i)

Which behavioural factors have influcences on the performance of RSPs? ii) How can these behavioural factors be captured? and iii) What are the strategies to optimise the performance of RSPs on parallel platforms?

The first sub-question involves identifying behavioural factors that have influences on the performance of RSPs. This sub-question will be addressed by Chapter 5 which includes quantitative analysis of the performance in terms of throughput and latency of RSPs. The performance analysis in Chapter 5 will help to determine the behavioural factors that affect the throughput and latency of RSPs.

The second sub-question is to identify techniques to capture these behaviour factors. Chapter 6 will introduce a monitoring framework which addresses this sub-question.

The third sub-question is to find strategies to optimise the performance of RSPs on parallel platforms. This sub-question will be tackled in Chapter 7 and Chapter 8. Targeting uniform shared-memory platforms, two novel approaches of using the properties of RSPs to derive scheduling priorities will be presented in Chapter 7. The first approach utilises the data demands on the stream communications while the second one makes use of the structural position of tasks in the RSP. These approaches support general RSPs especially with variable behaviours and dynamic structures which is a challenge when using static scheduling based on formal constraints or probabilities. They are therefore applicable for a large range of applications. Finally, Chapter 8 will introduce the usage of graph partitioning to map RSPs onto heterogeneous distributed systems. This chapter will also present two new heuristic algorithms to optimise the throughput of RSPs on heterogeneous distributed systems. Since these are off-line approaches, they are restricted to RSPs with relatively stable behaviour and relatively static program structures. In cases where these properties are highly varied, the proposed algorithms can be used to find the initial mapping. An adaptation method is then required to repartition during runtime.

Chapter 2

Background

In this chapter, we provide some background in the context of this thesis — reactive stream programs (RSPs). This includes a brief introduction of data flow programming, stream programming, and reactive systems. We also present the conceptional execution model for RSPs, and give an example of this execution model which is used as the main experimental environment for later chapters.

2.1 Data Flow Programming

Data flow programming is a paradigm that structures programs as networks or graphs of operator nodes connected by data arcs. In contrast to the traditional control-flow model, an operator node can be executed as soon as its data become available on the input arcs. The very first models of data-flow programming were initiated in 1960s with two main motivations: i) graphically describing computer programs [Sut66]; and ii) modelling parallel computations of programs [KM66, RRB69, ET63, Mar66]. The rest of this section gives an overview of data-flow programming. To get a broader and deeper view of data-flow computing, there are good surveys such as [WP94, JHM04, Ste97].

Since the first data-flow programming models, several approaches have been developed in two main categories based on the granularity of the operator implementation. The first one is the fine-grained data-flow model (also called the pure data-flow model) in which every machine instruction is an operator node. This direction leads to the appearance of data-flow-based hardware architectures. Some representative examples of this group are the MIT data-flow supercomputer [Den80] and the data-driven machine 1 (DDMI) [Dav78]. Since the fine-grained data-flow processors perform poorly on sequential code and the overhead of token matching is high, this direction was adapted

to the threaded data-flow model which is a combination of data-flow and control-flow mechanisms [PT91, SRU98]. In this model, a subgraph with a low degree of parallelism is transformed into a single operator node to be executed sequentially.

The second category evolves as a compromise between the pure data-flow approach and the traditional control flow approach. The approach is called large-grained data-flow in which each operator node is a chunk of code and is activated by the availability of data. This approach allows the expression of a complex program in a natural way as a composition of simpler components. There is a significant number of data-flow models based on this approach. Some examples are TDFL [Wen75], VAL [ADA79] and Kahn networks [Kah74]. Lucid [AW77] and SISAL [MSA+85] are also well known for data-flow programming although they were initially developed for other reasons. Lucid was originally designed as functional language with single assignment semantics to enable mathematical proofs of assertions about the program. Aiming for high level parallel programming, SISAL was developed as a structured functional language with single assignment semantics, implicit parallelism, and efficient array handling.

2.2 Stream Programming

2.2.1 Stream

Lee and Parks observed different definitions of the term *stream* in literature [LP95]. There are two main groups. Starting with the first group, Ladin defines streams in a recursive manner [Lan65]. A stream is defined by two components: one is the first element in the stream and the other is the procedure to compute the rest of stream. With this kind of definition, streams are usually treated with lazy semantics, i.e. elements on a stream only need to be produced when its consumer needs to process them [Bur75].

The second group defines streams as channels, or (possibly infinite) sequences of elements [FFJ90, Den95]. A stream is modified by adding or removing elements. In the context of this thesis, the latter definition is used, i.e. streams are regarded as channels.

2.2.2 Stream-Programming Model

Stream programming is a form of data-flow programming where streams are used to represent the data arcs. A stream program is structured by a set of operator nodes connected via streams. Following the data-flow model, operator nodes in a stream program communicate with each other via streams and they can only be executed as

soon as data in their input streams are available. For convenience, from now on we refer to operator nodes simply as *nodes*.

There are numerous projects investigating this approach. A good survey can be found in [Ste97]. This section provides an overview with some notable examples like Kahn Network, Synchronous Data Flow, StreamIt and S-Net.

In Kahn process networks (KPN), operator nodes are deterministic sequential processes, and data arcs are FIFO channels (streams) with unbounded capacities [Kah74]. This results in the deterministic behaviour of the whole network. A reading operation is blocking, i.e. a process becomes *blocked* when reading from an empty input stream. Since streams come with unbounded capacities, writing operations are non-blocking. Kahn proposed this mathematical model with the main intention to model concurrent systems. Later this model was found to be suitable for modelling signal processing systems.

Synchronous Data Flow (SDF) is a restricted variant of KPNs [LM87a]. When a node in SDF is invoked, it consumes/produces a fixed number of elements from each of its input/output streams. Each stream in SDF has a property called *delay*, which defines the data processing offset between the consumer and producer of the stream [LM87b]. If there is a delay x on a stream from node A to node B, the first x elements on the stream are not produced by A but are part of the initial state of the program. That means the (n)th element that B consumes is the $(n-x)$th element produced by A. With this constraint, SDF does not require unbounded channels as KPNs and it is guaranteed to have a static schedule.

As a more recent approach, StreamIt comes as a comprehensive framework including programming language, compiler and runtime system [TKA02, Thi09]. StreamIt employs the SDF model in which every node per invocation consumes and produces a given number of elements. To support node synchronisation, StreamIt introduces the conception of *information flow* where messages can carry timing information when transferred over streams [TKG+01]. The timing information can be used to specify the execution dependency between two nodes. For example, given a node A that communicates with node B via a stream s, A can only proceed if there are at least x messages in stream s. The timing information can also be used to describe the latency to be transferred from the sender node to the receiver node.

Developed in a different way than StreamIt, S-Net aims to support the transition from sequential code to parallel code as the concurrency handling is completely managed by S-Net [GSS08]. Nodes in S-Net are implemented by an independent computational language, e.g. ISO C. Nodes communicate over streams with typed messages. S-Net focuses

on coordination, i.e. how nodes connect to each other. S-Net programs are constructed hierarchically by combinators, such as parallel composition, serial composition, parallel replication and serial replication. Compared to StreamIt, S-Net is more general and closer to KPNs. In S-Net, there is no constraint about how many messages a node can consume or produce for one invocation. S-Net is asynchronous, i.e. there is no global clock for node executions across the network.

In addition, the program structure can be changed dynamically at runtime.

2.2.3 Properties of Stream Programs

Property	*Possible values*	
Stream Communication Type	Uni-directional	Bi-directional
Node Computation Type	Functional	Non-functional
Node Computation Behaviour	Constant	Variable
Inter-node communication	Synchronous	Asynchronous
Program Structure	Static	Dynamic

Table 2.1: Properties of stream programs

Stream programs can be classified by different properties of their nodes and streams [Ste97]. These properties are summarised in Table 2.1. In general, streams can be uni-directional or bi-directional. On a uni-directional stream, messages are transferred in one direction, i.e. only one task reads messages from the stream and another task writes messages to the stream. In contrast, a bi-directional stream can transfer messages in two directions, i.e. it connects two tasks and both of them can read messages to and write messages to the stream.

A node's computation can be a functional or non-functional program. The output messages of a functional node depends only on the input messages, i.e. a functional node produces the same output messages for the same input messages. For a non-functional node, the output messages depend not only on the input messages but also other factors, e.g. the internal state of the node.

The node's behaviour in terms of execution time and multiplexity can be variable or constant. For example, the execution time of a node can be constant for all input messages or can be varied depending on the value of the input message. Multiplexity of a node is the ratio of the number of input messages to the number of output messages per node invocation. For example, whenever a node with multiplexity of n-*to*-m is invoked, it reads in n input messages and writes out m output messages. A node with constant multiplexity has n and m unchanged for all invocations. In contrast, various

multiplexity indicates that the value of n and m can be different for each invocation. They can depend on the internal state of the node or the value of input messages.

The inter-node communication within the stream program can be synchronous or asynchronous. Within a synchronous stream program, each stream communication between a pair of nodes requires a clock-like mechanism to synchronise the transmitting and receiving messages. For example, in SDF programs, each stream has a property called `delay`, which defines the data processing offset between the producer and consumer of the stream. In asynchronous stream programs, there is no notion of time and the message reading and writing via a stream are proceeded independently. For example, each stream in an S-Net program can be written to at any time and can be read from at any time as long as it contains messages.

In addition, the program structure in terms of nodes and their stream connections can be dynamic or static, i.e. it may change dynamically or remain fixed during the program's lifetime. Although dynamic structures are not mentioned in [Ste97] we consider this an important factor in scheduling stream programs.

2.3 Interactive/Reactive Systems

The conception of reactive systems was first introduced in 1985 by Harel and Pnueli to differentiate with the traditional view of computer programs [HP85]. In the traditional view, a computer program is considered as a black box accepting inputs, performing transformations, returning outputs and terminating. Such programs are called transformational systems and they can be completely described as relations between their inputs and outputs. In contrast, a reactive system interacts continuously with the environment to maintain an ongoing relationship between them. The reactive system and the environment are connected via the input and output interface. A reactive system cannot be completely described only by the relation between inputs and outputs. The sufficient description of a reactive system must refer to the ongoing sequence of system's states [Pnu86].

Based on the relationship between the system and the environment, Berry splits reactive systems into two categories: reactive systems and interactive systems. Reactive systems are those that react continuously to their environment at the speed of the environment, in contrast with interactive systems which react with the environment at their own speed [Ber89]. Reactive systems can be real-time, e.g. air-bag systems in cars; or non real-time, e.g. communication protocols. This distinction is used in most of literature e.g. [Hal98, ELLSV99].

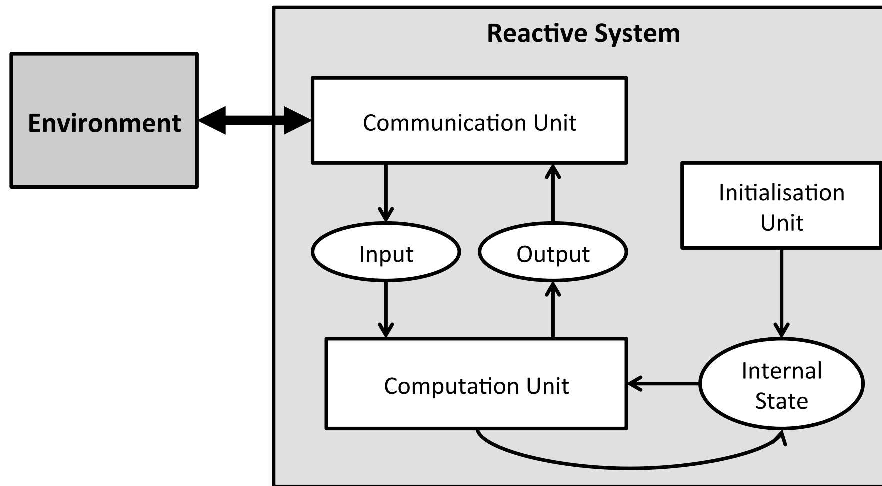


Figure 2.1: Abstract structure of interactive/reactive Systems

Schneider later gives a more distinguishable view point in [Sch04]. When dealing with an interactive system, the environment has to wait for the system to be ready for new inputs. In contrast, the environment of a reactive system can decide when to initialise the interaction, i.e. the environment is free to send new inputs at any time. In addition, Schneider argues that reactive systems have to be fast enough to satisfy time constraints from the environment and therefore they are real-time systems. This contradicts from the definition of Berry as presented above. Although we appreciate the new distinguished point of Schneider, we don't use their argument that reactive systems automatically have to be real-time systems, as our work does not explicitly focus on real-time computing.

Although interactive and reactive systems have different ways of reacting to the environment, they have a similar structure. Figure 2.1 shows an abstract description of interactive/reactive systems which includes three main units and an internal state. The initialisation unit is used to set the first value for the internal state. The communication unit receives inputs from the environment and passes them to the computational unit. The computational unit processes the data depending on the behaviour controls derived from the internal state. After processing the data, the computation unit updates the internal state and also passes the outputs to the communication unit to return to the environment.

2.4 Context of this Thesis: Reactive Stream Programs

As discussed in the previous section, reactive and interactive systems are distinguished by their ways of reacting to the environment. This behaviour is part of the application's semantics and there is no structural difference between reactive and interactive systems.

For this reason, within this thesis, we disregard any distinction between interactive and reactive systems. Instead we adhere to the original definition of reactive system given in [HP85], and see reactive systems as being interactive.

This thesis focuses on best-effort scheduling strategies for RSPs. An RSP is defined as a reactive system with its internal implementation designed as a stream program. Scheduling strategies presented in this thesis aim to optimise the performance by using stream-related features of the programs.

The work of this thesis targets general RSPs with properties shown in Table 2.2. In these RSPs, the node computation can be functional or non-functional, and the node computation behaviour can be constant or variable. The inter-node communication can be synchronous or asynchronous, and the program structure can be static or dynamic. The work in this thesis is restricted to RSPs with uni-directional streams. However, a bi-directional stream can be modelled as a pair of uni-directional streams, one for each direction. Since in reactive systems there is no strong reason to restrict system implementations to a concrete type of node computation, node behaviour, inter-node communication, or program structure. Thus we have chosen to target general RSPs in order to have a wide applicability of the research results.

The scheduling strategies in Chapter 7 are applicable for general RSPs while those in Chapter 8 are more restricted. In particular, they are only applicable to RSPs with relative constant node computational behaviour and relative static program structures. That means these properties are not highly variable during the runtime.

Property	*Possible values*	
Stream Communication Type	Uni-directional	
Node Computation Type	Functional	Non-functional
Node Computation Behaviour	Constant	Variable
Inter-node communication	Synchronous	Asynchronous
Program Structure	Static	Dynamic

TABLE 2.2: Properties of RSPs supported in the dissertation

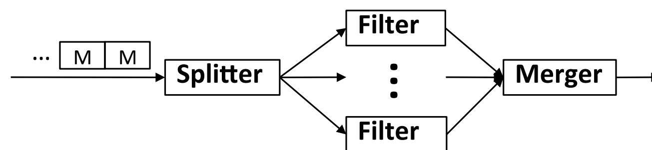


FIGURE 2.2: Example: program structure of an image filter application

An example of an RSP is shown in Figure 2.2. This is the structure of an image filter application. The RSP includes a node *Splitter*, which reads in images and splits each of them into sub-images. All sub-images are scattered in different branches where nodes

of type *Filter* apply the filter on each sub-image. The filtered sub-images are then sent to the node *Merger*. *Merger* unifies them into a complete image and sends it out.

2.4.1 Data in Reactive Stream Programs

In RSPs, data arrives from the environment as a virtually infinite sequence of messages. Nodes that receive messages from the environment are *entry nodes*. Nodes sending messages to the environment are *exit nodes*. In the example of Figure 2.2, *Splitter* is an entry node and *Merger* is an exit node. An RSP can have multiple entry nodes and multiple exit nodes. Input messages of entry nodes come from the external environment and are therefore called *external input messages*. Similarly, output messages of exit nodes are called *external output messages*. Other messages inside the RSP are referred to as *intermediate messages*.

2.4.2 Execution Model of RSPs

Conceptually the execution model of RSPs includes three layers: a compiler, a runtime system (RTS) and an execution layer. The source code of an RSP is first passed through the compiler to generate the object code. The RTS uses the object code to allocate runtime objects including FIFO buffers and runtime components. Each FIFO buffer represents a stream and each runtime component (RC) represents an instance of a node. In some RTSs, for example S-Net, a runtime component can represent a node or an operator. The behaviour of these operators is similar to that of nodes, i.e. they read input messages, process them and write output messages. Unlike nodes, an operator does not involve itself in computational activities but instead participates in controlling the behaviour of the RSP. For example, an operator can direct messages to the appropriate streams, or it can dynamically change the structure of the RSP.

The duty of the RTS is to enforce the RSP's semantics. The RTS maintains the graph of RCs, i.e. makes sure that each runtime component is connected to the appropriate streams. As shown in Table 2.1 (on page 14), the structure of RSPs can be dynamic. The set of runtime components and their stream connections can be dynamically changed during runtime. The RTS is therefore responsible for applying these structural changes.

Beneath the RTS, the execution layer provides a mechanism to wrap RCs into executable objects called tasks. A task is a process that repeats **RC invocations**. For each RC invocation, a task gathers n messages from a set of input streams. The task then processes these n messages to produce m output messages. These m output messages are then scattered to a set of output streams. n-*to*-m is called the multiplexity of the

task, and the execution of the RC invocation can occur only when n required input messages are available. As the node's behaviour can be non-deterministic, the two values n and m can be unpredictable. Also m produced messages can be dynamically distributed to the output streams. In the example in Figure 2.2, node *Splitter* can split one image into a number of sub-images depending on the image size. Depending on the implementation, this node can have a policy dictating how to scatter these sub-images into different branches. For example, one can choose to scatter in a round-robin manner.

Besides task management, the execution layer also includes the implementation of FIFO buffers to transfer messages between tasks. A **stream transfer** of a message M is defined as the activity of moving M across a stream from one task to another. The activity of a stream S therefore consists of stream transfers of all messages passing over S. On shared memory platforms, message transference is usually implemented simply by memory access operations, while on distributed platforms it is implemented by message passing, e.g. Message Passing Interface (MPI).

The graph of RCs now becomes the graph of tasks so-called *task graph.* Tasks associated with entry nodes are called *entry tasks.* Tasks associated with exit nodes are called *exit tasks.* Other tasks are called *middle tasks.*

The execution layer also controls the state of tasks, i.e. when a task is ready to be scheduled. According to the model of KPN, streams are unbounded and a task is ready to be executed if all required messages are available on their input streams. In this case, the task state is *ready.* Otherwise the task state is *blocked.* To avoid the overloading problem and support back pressure, some execution models implement streams as bounded buffers which results in additional scheduling constraints. With bounded buffers, a task can be blocked when trying to write to a full stream.

As the most important part of the execution layer, the scheduler employs a policy to execute ready tasks on a platform consisting of physical resources. The scheduler's policy decides:

- which ready task will be executed;
- which physical resource will perform the ready task; and
- the length of scheduling cycle, i.e. the period for which the physical resource will perform the ready task

Each physical resource can be a CPU core on shared memory platforms, or a processing element (PE) on distributed platforms. In the latter case, the scheduler can be hierarchical where it includes a mapper to distribute tasks onto PEs, and a local scheduler for time-shared scheduling within each PE.

2.4.3 Message Derivation

When an RC invocation consumes an input message M_x (and possibly other messages) to produce an output message M_y (and possibly other messages), it is said that M_x derives M_y, or M_y is derived from M_x, formally written as $Drv(M_x, M_y)$. In this case, M_x is a *predecessor* of M_y and M_y is a *successor* of M_x. The message derivation relation is transitive, i.e. $Drv(M_x, M_y) \wedge Drv(M_y, M_z) \implies Drv(M_x, M_z)$. In this case, M_y is directly derived from M_x and M_z is indirectly derived from M_x. To distinguish these two types of derivations, we use *DDrv* for direct derivation and *IDrv* for indirect derivation. We now have the following implications.

$$DDrv(M_x, M_y) \implies Drv(M_x, M_y) \tag{2.1}$$

$$IDrv(M_x, M_y) \implies Drv(M_x, M_y) \tag{2.2}$$

$$Drv(M_x, M_y) \implies IDrv(M_x, M_y) \vee DDrv(M_x, M_y) \tag{2.3}$$

$$Drv(M_x, M_y) \wedge Drv(M_y, M_z) \implies Drv(M_x, M_z) \tag{2.4}$$

$$DDrv(M_x, M_y) \implies \neg IDrv(M_x, M_y) \tag{2.5}$$

$$IDrv(M_x, M_y) \implies \neg DDrv(M_x, M_y) \tag{2.6}$$

The derivation relation between messages can be used to form a directed graph whose nodes represent messages and edges reflect the derivation relations. There is a directed edge from message M_x to message M_y if there exists $DDrv(M_x, M_y)$. This graph is called a **Message Derivation Graph (*MDG*)**. Vertices of the MDG represent *message nodes*. Figure 2.3 shows an example of message derivation from external input messages to external output messages. In this example, we have from external input I_2 towards external output O_3 the derivations $DDrv(I_2, M_1)$, $DDrv(M_1, M_4)$ and $DDrv(M_4, O_3)$. We also have $DDrv(M_1, M_5)$, $DDrv(I_3, M_5)$, and $DDrv(M_5, O_4)$, etc.

For a message M, we denote *d_successor*(M) as the set of messages that are directly derived from M; *i_successor*(M) as the set of messages that are indirectly derived from M; and *successor*(M) as the set of messages that are either directly or indirectly derived from M. Similarly, we have *d_predecessor*(M) and *i_predecessor*(M) which are the set

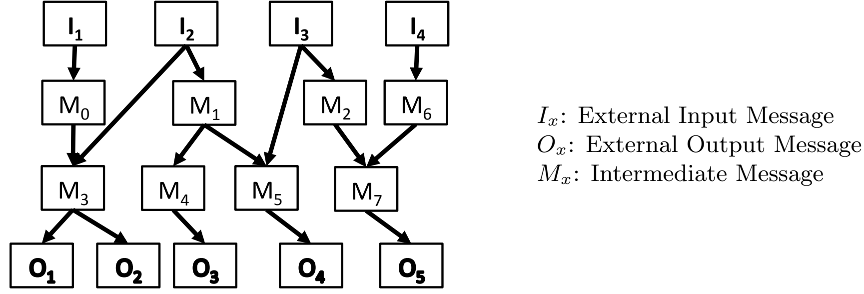


FIGURE 2.3: An example of message derivation

of its messages that directly and indirectly derive M. Also $predeccessor(M)$ is the set of messages that either directly or indirectly derive M. The formal definitions of these sets are:

$$d_successor(M) = \{M_x \mid DDrv(M, M_x)\} \tag{2.7}$$

$$i_successor(M) = \{M_x \mid IDrv(M, M_x)\} \tag{2.8}$$

$$successor(M) = d_successor(M) \cup i_successor(M) \tag{2.9}$$

$$d_predecessor(M) = \{M_x \mid DDrv(M_x, M)\} \tag{2.10}$$

$$i_predecessor(M) = \{M_x \mid IDrv(M_x, M)\} \tag{2.11}$$

$$successor(M) = d_predecessor(M) \cup i_predecessor(M) \tag{2.12}$$

A message M is an external input message if there is no message that derives M, i.e. $predecessor(M) = \emptyset$. Similarly, M is an external output message if it does not derive any other message, i.e. $successor(M) = \emptyset$. We define $derived_output(M)$ as the set of external messages that are directly or indirectly derived from M respectively. This includes all leaves of the tree rooted by message node M.

$$derived_output(M) = \{M_x \mid M_x \in successor(M) \wedge successor(M_x) = \emptyset\} \tag{2.13}$$

2.4.4 Message Completion

An external input message I_i when processed by an RSP may derive multiple intermediate messages M_j before deriving any external output messages O_k, written as $DDrv(I_i, M_j)$, $DDrv(M_j, O_k)$ for each such M_j. An external input message I_i is said to be **completed**, i.e. completely processed, when all of its derived external output messages $derived_output(I_i)$ have been sent out. In the example of Figure 2.3, the external input message I_2 is completed when the messages O_1, O_2, O_3, O_4 are all sent out.

The **completion of an external message** is defined as a set of node invocations that produce all its successor messages and the stream transfer of these messages. Put in another way, completion of an external input message I_i is the process of performing RC invocations to generate messages in $successor(I_i)$, and stream transfers to pass them to other task until all its derived external output messages $derived_output(I_i)$ are sent out.

As mentioned above, a task executes a sequence of RC invocations. The **contribution** of a task T to an external input message I_i is defined as the group of T's RC invocations that belongs to the completion of I_i. Similarly, **contribution** of a stream S to an external input message I_i is the group of S's stream transfer that belongs to the completion of I_i.

2.5 Instantiation of the Execution Model for Reactive Stream Programs

2.5.1 Stream programming with S-Net

S-Net [GSS08] is a declarative coordination language that aims to separate computations from concurrency management aspects. The computational logic is meant to be encapsulated inside the individual computational components, also called *boxes*, while S-Net focuses on how to connect the communication between these components via streams. For this reason, S-Net fits stream applications that process long sequences of data well. Figure 2.4 shows the S-Net code of the Image Filter Application (described in Section 2.4).

```
net ImageFilter ({Img} -> {Img})
{
  box Splitter((Img) -> (SubImg, <branch>));
  box Filter((SubImg) -> (SubImg));
  box Merger((SubImg, <branch>) -> (Img));

} connect Splitter .. Filter!<branch> .. Merger;
```

FIGURE 2.4: S-Net implementation of the image filter application

Boxes in an S-Net program are re-entrant procedures (written in a conventional programming language) that transform a message from a single input stream into a sequence of messages on a single output stream without any persistent state, i.e. the value of none of the internal variables is preserved from one execution to the next.

Each message in an S-Net program is comprised of a set of entries, each of which is represented by a label. The set of labels defines the type of the message. There are two different kinds of entries: fields and tags. The data of fields is not revealed to the S-Net RTS, while tags are integers which are recognised by both the S-Net RTS and within the user-defined boxes. In S-Net, tags are enclosed within angular brackets. In the above example, there are three different types of messages: (Img), $(SubImg, \langle branch \rangle)$ and $(SubImg)$. $\langle branch \rangle$ is a tag while others are fields.

In order to construct an S-Net program composed from boxes, S-Net provides four combinators. The first two are static combinators, named serial composition (denoted as `..`) and parallel composition (denoted as `|`), to construct pipelines and branches respectively. They are static in the sense that only one instance for each of their operands is created. The other two combinators are dynamic in the sense that they create replicas of their operands on demand by means of serial and parallel replication. Serial replication (named the $\star$-combinator in S-Net) are used to instantiate execution pipelines of dynamic lengths. Messages in this pipeline are processed and forwarded to the next stage until the specified exit condition is met. Parallel replication (named the !-combinator) creates a dynamic number of instances of its operand and combines them in parallel. Each message is processed by only one of these instances; the concrete instance is determined by a tag value that the message is expected to carry. Among the above combinators, only serial compositions preserve message order while others do not. To support this, S-Net provides for each of them a special order-preserving variant (denoted as `||`, $\star\star$, and `!!`)

The declaration of each S-Net box includes one input type, i.e. the type of message that the box accepts as input; and multiple output types, i.e types of messages that the box produces as output. To loosen the restriction of a single input type, S-Net support an inheritance mechanism, called *flow inheritance*. This allows a box to accept all sub-types of the box's declared input type. Excess fields and tags of a message are bypassed through the box. That means when a message arrives to a box, only entries with listed labels in the input type are taken by the box to generate output messages. Other excess entries are passed over the box and added to each output message. In the case where an output message contains an entry with the same label, the bypassed label will be discarded. In the example in Figure 2.4, the output messages of $Splitter$ with type $(SubImg, \langle branch \rangle)$ are sent to $Filter$ whose input type is $(SubImg)$. Only fields labelled $SubImg$ are consumed by $Filter$, while tags labelled $\langle branch \rangle$ are bypassed and added to the output messages which are sent to $Merger$.

Boxes in S-Net are SISO entities, i.e. each box has a **S**ingle **I**nput stream and a **S**ingle **O**utput stream. Therefore, if a box requires data from several messages as input, these

messages have to be merged first. S-Net provides a primitive entity for this purpose, called *synchro-cell*. A synchro-cell is parameterised over the type of message that it is supposed to merge. As soon as it receives messages of all matching types, it releases a single combination of these messages.

Streams in S-Net are used to connect entities. They are uni-directional and operate in the FIFO manner. Each stream has a single reader and single writer. An S-Net program can be described as a network of entities connected by streams.

2.5.2 S-Net Compiler and Runtime System

The execution model of S-Net is shown in Figure 2.5. An RSP written in S-Net is passed to a compiler and translated into an internal representation. Based on this internal representation, the compiler performs basic consistency checks and applies optimisations. Finally, the compiler generates C-code in a format called the *common runtime interface* (CRI) [GP11].

Lying under the compiler, the S-Net runtime system (RTS) includes a CRI deployer which takes the CRI code and produces a graph of runtime components (RCs) connected by streams. The graph of RCs, also called the RC graph, is a derived representation of the original S-Net program. In fact, each RC represents an S-Net entity or operator. S-Net operators are created to provide desired behaviours of S-Net combinators. Serial composition is the simple case where operands are connected by pipelined streams, and therefore no extra operator is required. Figure 2.6 shows the runtime operators represented for parallel composition, parallel replication and serial replication. To distinguish entities and operators, figures of RC graphs will denote operators in a pair of angular brackets <>.

A parallel composition is represented by a pair of operators: parallel compositor and collector. The former distributes messages from its input stream to operand branches while the later gathers output messages of these operand branches. A similar design is used for parallel replications, except that the parallel replicator also generates new operand instance when an input message comes with a tag different from all existing operand instances. Figure 2.6b shows the parallel replication with one generated operand.

Each serial replication starts with a serial replicator and a collector. When receiving a message, the serial replicator passes it to the collector if it matches the exit condition. Otherwise, the message will be sent to the next operand instance. The next operand instance will need to be generated if it does not exist. A new serial replicator is also generated following the new operand instance as shown in Figure 2.6c.

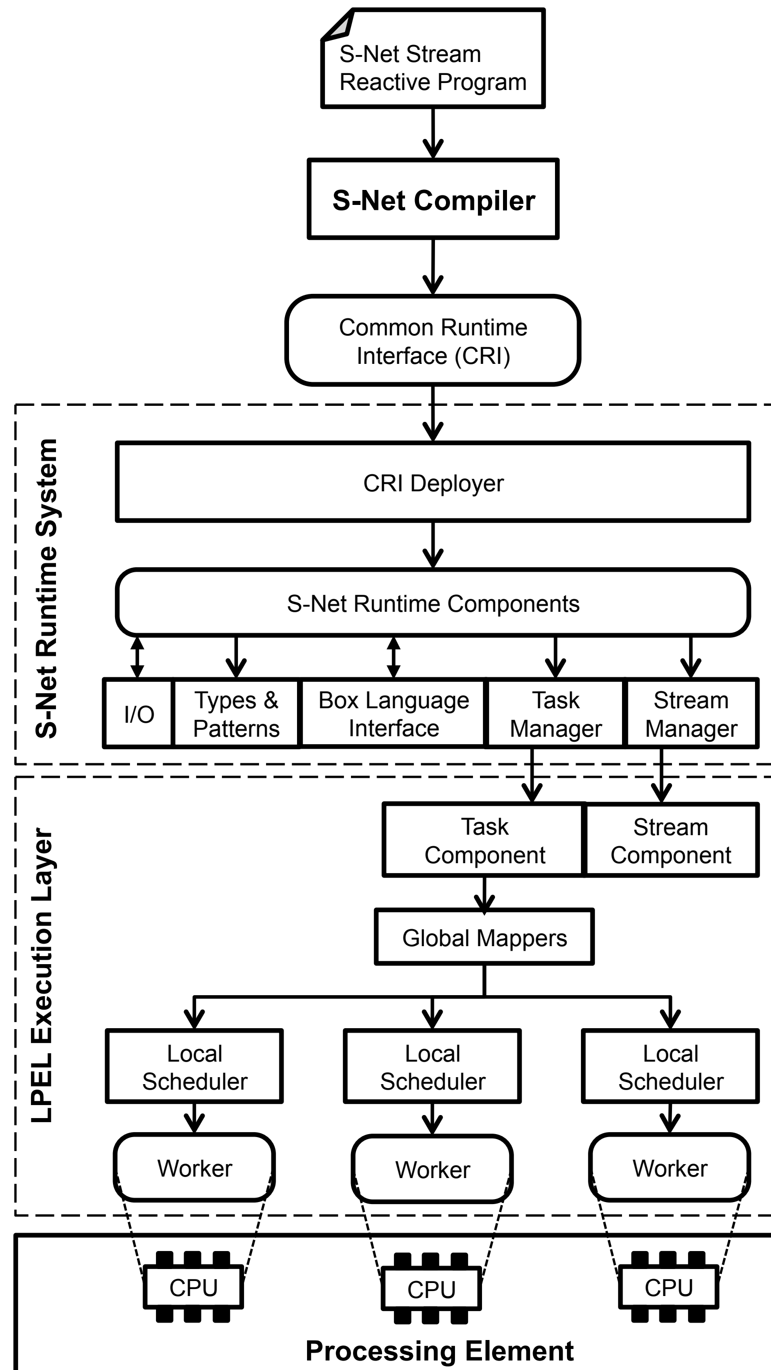
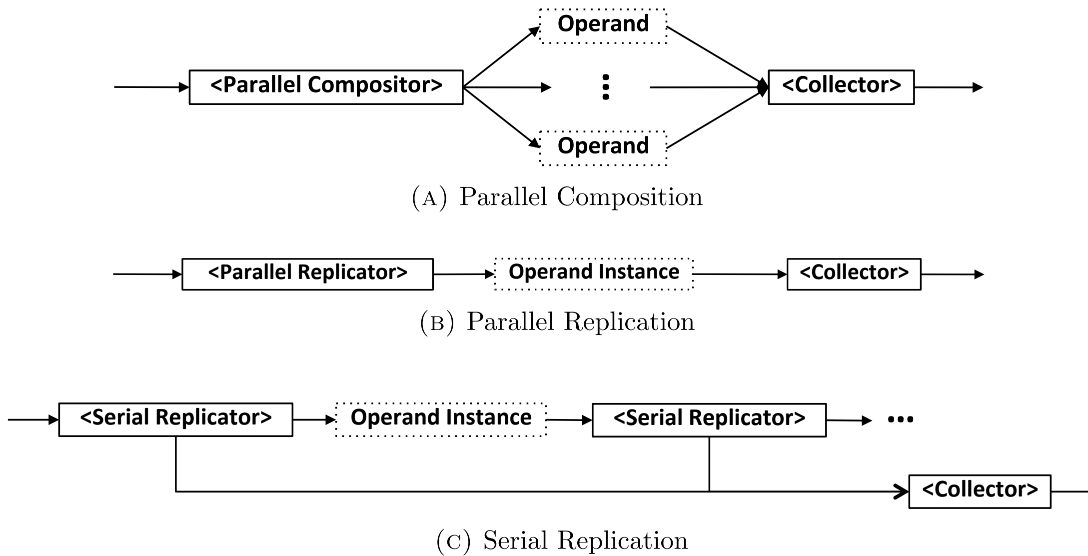


FIGURE 2.5: An overview of the S-Net execution model using LPEL

At the runtime system level, each stream is represented as a FIFO buffer with a single reader and single writer. Since S-Net streams are uni-directional and no operation in S-Net allows a stream to connect backward to the previous entities, the RC graph is directed and acyclic. Figure 2.7 shows the RC graph of the Image Filter Application with 3 instances of box *Filter*.

Apart from the CRI deployer, the S-Net runtime system also includes modules to support types and patterns; I/O communication and interfacing with the box language. In



(A) Parallel Composition

(B) Parallel Replication

(C) Serial Replication

FIGURE 2.6: Runtime expansion of network combinators in S-Net

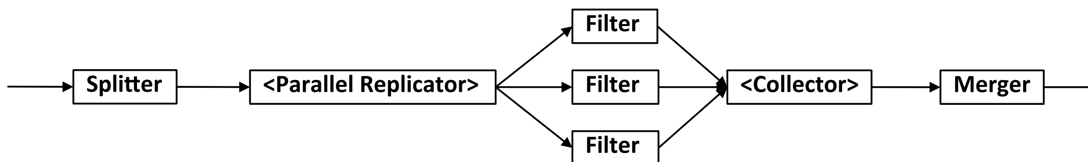


FIGURE 2.7: The runtime component network of the image filter application (with three instances of box *Filter*)

addition, there is a task manager to facilitate task creation and destruction. Similarly, a stream manager is provided for stream creation and destruction.

2.5.3 LPEL — A User-mode Microkernel for the Streaming Language S-Net

The Light-Weight Parallel Execution Layer (LPEL) [Pro10] is an execution layer designed for S-Net to give control over mapping and scheduling. LPEL adopts a user-level threading scheme providing the necessary threading and communication mechanisms in user-space. It builds upon the services provided by the operating system or virtual hardware, such as kernel-level threading, context switching in user-space, atomic instructions and timestamping.

On LPEL, there is a stream component to support stream creation, stream reading, stream writing and stream replacing. Additionally, a task component is provided to create a wrapper around each RC before sending them to the scheduler. At the LPEL level, each S-Net RC is wrapped in a user-level thread, called a *task*. As there is an one-to-one mapping between tasks and RCs, they are used interchangeably. Each task is an iteration of RC invocations, each of which performs either one box's computation,

one synchronisation for a synchro-cell, or one activity of a operator. The activity of an operator is to read one message and guide it to the appropriate stream. In the case of a serial or parallel replicator, the activity can include creating a new operand. For example, a parallel replicator's activity is to read in a message, choose the branch with the correct tag, create the operand branch if it does not exist, and send the message to the branch via the connecting stream.

Tasks communicate with each other via streams. Each stream is a uni-directional communication channel between two tasks.

LPEL is designed to execute S-Net programs on shared memory platforms. Tasks are distributed on workers, each of which represents a CPU core or a hardware thread. The scheduling policy determines a task with a *ready* state to be dispatched on a worker. The state of a task changes according to the availability of the input streams. Reading from an empty stream causes the task to be blocked, and writing to an empty stream can unblock the task on the other side of the stream.

The current scheduler of LPEL distinguishes between two types of tasks: box tasks associated with computational boxes in S-Net; and non-box tasks associated with S-Net operators which contain no computation. The LPEL scheduler employs two global mappers: one for box tasks and one for non-box tasks. When a task is created, depending on the task type (box or non-box) it is distributed among workers by one of the mappers. Both of the global mappers are implemented in a round-robin manner. Each worker has its own local scheduler which manages its set of assigned tasks and facilitates a round-robin scheduling policy. To avoid memory overloading and to create back pressure, LPEL requires streams to be bounded. That means writing to a full stream causes the writing task to be blocked. Likewise, reading from a full stream unblocks the task on the other side of the stream.

2.5.4 S-Net on Distributed Systems

To support stream programs on a distributed system of processing elements (PEs), the S-Net language is extended with a placement-annotation mechanism. This allows the programmer to statically or dynamically map S-Net RCs to different PEs. In particular, @NUM is used to statically map a given set of RCs to the PE indexed NUM. For example, $(A..B)$@1 indicates that both boxes A and B are mapped to the PE indexed 1; and $(A|B)$@2 indicates that box A, box B, the parallel compositor and the collector are mapped to the PE indexed 2. Dynamic placement is only supported for the parallel replicator by using @ before its tag value. For example, $(A..B)$!@ $< tag >$ shows that when a branch of $A..B$ is created, it will be mapped to the PE with the same index as

its tag value. In the case where the placement of an RC is not annotated, the RC is mapped to the default PE with index 0.

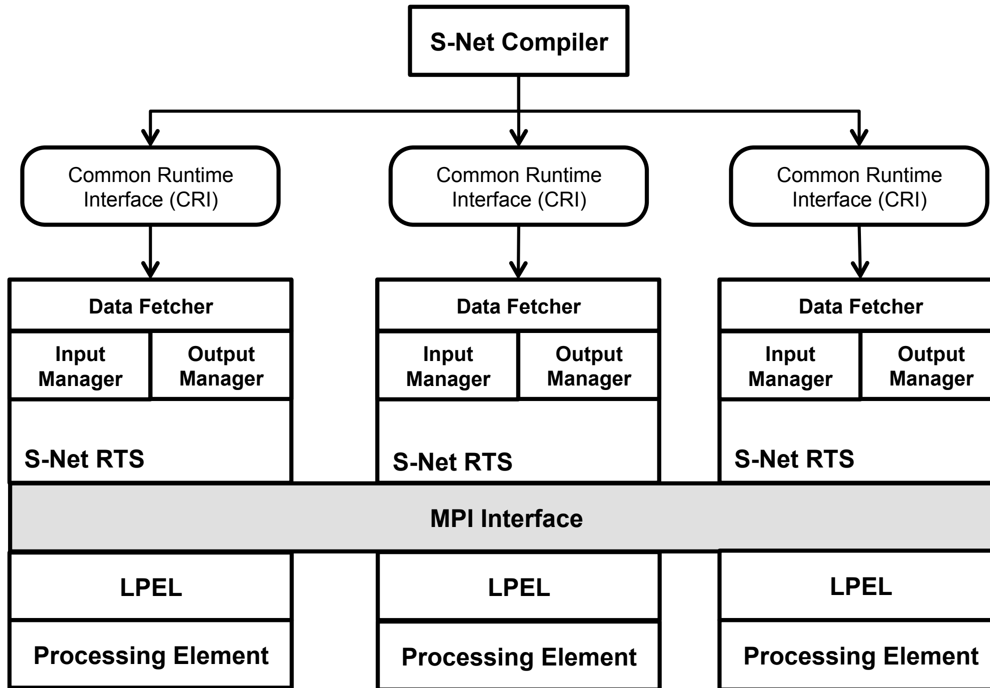


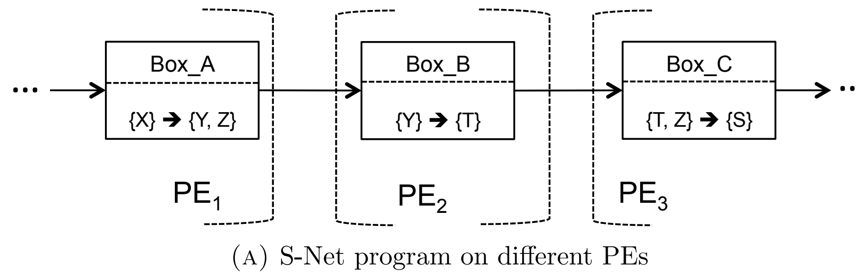
FIGURE 2.8: Execution model of distributed S-Net

To support S-Net on distributed systems, each PE is equipped with its own S-Net RST and LPEL execution layer. Figure 2.8 shows the execution model for distributed S-Net. The compiler first takes the S-Net program with placement annotations and generate the CRI code for each PE. Within each PE, the CRI code is used to create LPEL tasks and streams. LPEL is employed for each PE and controls the scheduling of tasks within the PE.

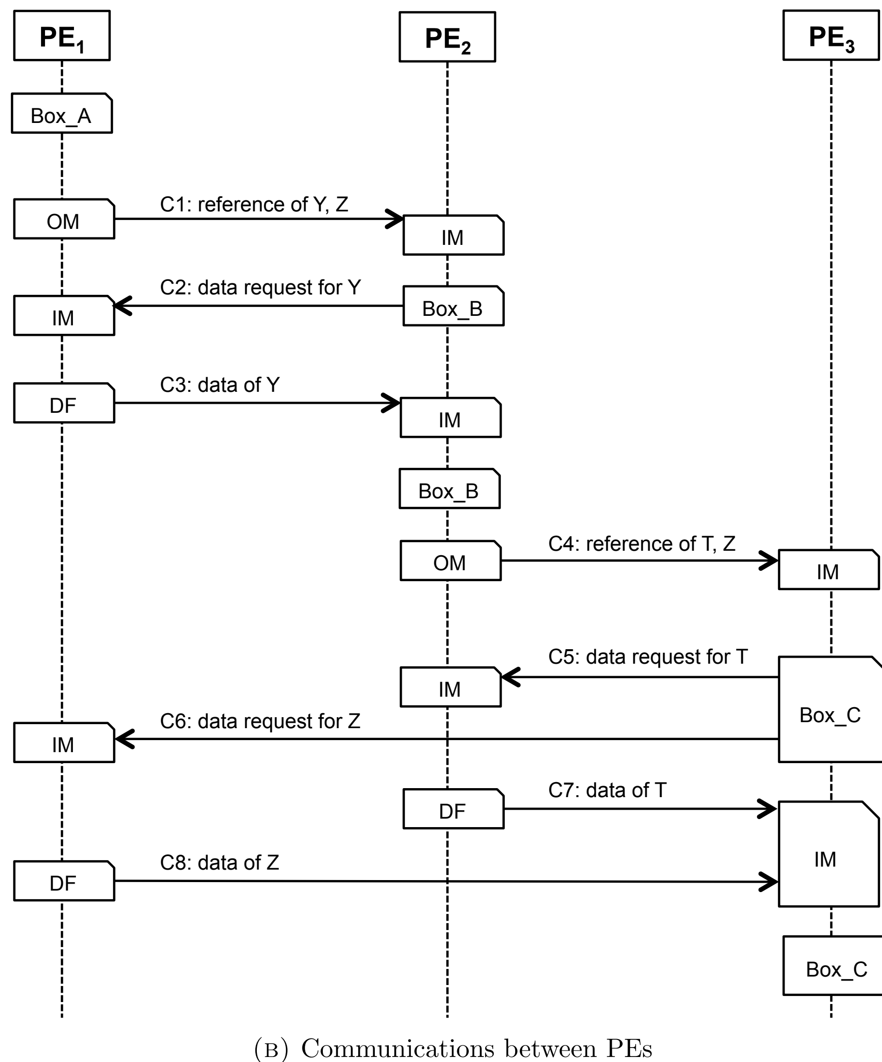
To maintain the stream communications across the border of PEs, each S-Net RTS employs two data managers: the *input manager* (IM) and the *output manager* (OM). The out manager keeps the set of streams, each of which is written by one RC at the current PE, and is read by another RC at another PE. The OM reads messages from these streams and uses the MPI interface to send them to the corresponding PE. Similarly, the IM manipulates the set of streams that are read by RCs at the current PE, and are written by RCs at other PEs. The IM receives messages by the MPI interface and writes to the corresponding streams.

To avoid unnecessary data transfers, the S-Net RTS represents each field of a message by a *reference*, which consists of a unique data identifier (UID) and the PE where the data is held. The relationship between the data and its UID is managed by a special component, called the *data fetcher* (DF). While transferred via the MPI interface, only the references of fields are sent. When the data is actually needed by an RC, the RC will

send a fetch request to the IM of the PE where the data is kept. The IM then informs the DF of the same PE. This DF retrieves the data and sends it to the responding PE.



(A) S-Net program on different PEs



(B) Communications between PEs

FIGURE 2.9: Example of an S-Net program being deployed on a distributed system

This is an efficient design to support flow inheritance and to implement S-Net operators where the data within each field is not directly retrieved. Figure 2.9a shows an example where three boxes *Box_A*, *Box_B* and *Box_C* of an S-Net program are mapped onto three different PEs. The interaction among these PEs is shown in Figure 2.9b. Although *Box_B* only needs the data of field Y from *Box_A*, PE_1 and PE_2 need three PE-to-PE

communications. On PE_1, *Box_A* is executed writing its output of (Y, Z) to its output stream, the OM of PE_1 sends the message containing reference of Y and Z to PE_2 (C1). The IM from PE_2 receives the message and writes it to the input stream of *Box_B*. *Box_B* receives the reference of both Y and Z but needs the data of Y only. *Box_B* then sends a request to PE_1 (C2). The IM of PE_1 passes the request to the DF of PE_1. The DF then sends the data of Y to the IM of PE_2 (C3). And finally the IM of PE_2 passes the data of Y to *Box_B* so that it can continue its computation. Due to the flow inheritance, the output message of *Box_B* consists of two fields T and Z. Their reference is then sent to *Box_C* in PE_3 (C4). *Box_C* requires the data of both Z and T. The data of Z is held in PE_1 while the data of T is held in PE_2. *Box_C* then sends the data request to the IM of both PE_1 and PE_2 (C5 and C6). The IM of PE_3 then receives the data of Z and T from the DF of PE_1 and PE_2 (C7 and C8). The data of Z and T is sent to *Box_C* to continue its execution.

The IM, OM and DF are implemented as separated threads and are not controlled by the LPEL scheduler. This design is necessary to avoid deadlocks and also to minimise the influence on concurrent task execution [GJP12].

2.6 Chapter Summary

This chapter provided the background of the context upon which of this thesis is based. The thesis focuses on best-effort scheduling strategies for RSPs. An RSP is a reactive system with its internal implementation designed as a stream program. As a reactive system, an RSP, as defined in [HP85], maintains an ongoing relationship with the external environment by receiving an infinite sequence of input messages and sending out an infinite sequence of output messages. Implemented as a stream program, an RSP is structured as a set of operator nodes connected via streams. A stream, as defined in [FFJ90] and [Den95], is a channel to transfer an infinite sequence of message between operator nodes. The chapter also introduced different properties of RSPs, and described the conceptual execution model of RSPs including the compiler, the runtime system and the execution layer.

In addition, the chapter made clear the class of RSPs which are supported by scheduling strategies in Chapter 7 and Chapter 8. In particular, Chapter 7 will propose scheduling techniques on uniform shared memory platforms. These scheduling techniques in Chapter 7 will require no assumption about the properties of the RSPs. They are therefore applicable for general RSPs, especially with variable node behaviour and dynamic

program structures. Chapter 8 will introduce mapping strategies of RSPs onto heterogeneous distributed systems. These strategies will only be applicable for RSPs with relatively stable node behaviour and relatively static program structures.

An instantiation of the execution model supporting S-Net stream languages was also included in this chapter. This execution model will be used as a benchmark platform for experiments to evaluate the proposed scheduling strategies later in Chapter 7 and Chapter 8.

Chapter 3

Related Work

Data-flow based stream processing has received a lot of attention in the context of scheduling. While implicit synchronisation makes it easier for the programmer, it burdens the system designer with providing efficient scheduling techniques. It is difficult to obtain good performance when the scheduler has no knowledge of internal behaviour of stream programs. To the best of our knowledge, our work in Chapter 6 is the first to propose a monitoring framework specialising in stream programs. In this chapter, we include an assortment of work in monitoring parallel programs where the internal activities are captured for the purpose of performance analysis. We also present primary types of scheduling algorithms, some of which are used in later chapters. In addition, we present a selection of work on scheduling stream programs classified by the target, i.e. many-core or distributed systems; and by the manner of the scheduling strategy, i.e. static or dynamic. Since we use graph partitioning for mapping stream programs onto distributed systems in Chapter 8, we include in this chapter some representative graph partitioning algorithms.

3.1 Monitoring Parallel Programs

To obtain the best performance of stream programs, it is necessary to understand their behaviour. For this reason, one focus of this thesis is to build a monitoring framework for parallel stream programs. To the best of our knowledge, our monitoring framework described in Chapter 6 is the first one specialising in analysing the performance of stream programs. As the closest related work we have identified monitoring of parallel programs, for which we describe a selection in this section. Unlike sequential programs where the performance is unchanged as long as the execution environment is stable,

the performance of parallel programs is heavily dependent on the way workload is distributed over the resources. In stream programming where tasks can be executed fully asynchronously, the scheduling strategies of the execution layer are as important for the performance as the internal execution of each individual task. We therefore focus here on approaches that capture the impact of scheduling strategies in performance analysis. We limit our view to automated tools that do not require explicit code instrumentation from the programmer.

One of the very early approaches is IPS which supports automated performance analysis of parallel and distributed programs [YM89, MCH+90]. Using knowledge of the program's structure, IPS aims to capture runtime information at five different levels ranging from primitive activities such as task entry and exit, blocking and unblocking by the scheduler, procedure level behaviour such as critical paths, process-level-based events such as inter and intra process communication, machine level information such as the summary of communications across machines, to general program behaviour such as the total execution time.

The Paradyn [MCC+95] framework provides a configuration language called the Paradyn Configuration Language (PCL) that allows the programmer to describe the desired performance metrics. Taking the requirement from the programmer, Paradyn dynamically adds the monitoring codes as instrumentations into the executable programs during execution. To control the overhead, Paradyn adjusts the monitoring behaviours depending on the cost of its data collection and a user-defined threshold. One usage of Paradyn is presented in [XLM97] to find the performance bottleneck of shared-memory parallel programs.

The TAU performance system provides a selection of tools categorised in three levels: instrumentation, measurement and analysis [SM06]. The framework can be configured to combine different tools from these three levels to satisfy different customised performance targets. The framework supports both automatic instrumentation and an API for the programmer to manually annotate the source code of the program. The TAU framework has been integrated into various systems such as the performance measurement infrastructure for Common Component Architecture (CCA) [MST+05], the online performance monitoring framework SuperMon [NSM+07], the global performance monitoring framework for MPI [HMSM06], etc.

Targeting large-scale HPC programs, the Scalasca toolset [GWW+10] captures concurrent behaviour of the program and uses tracing techniques to aggregate performance metrics. The toolset particularly focuses on its scalability as well as the integration capabilities to transform raw measurements into the execution behaviour description and performance summarisation.

In contrast to most of monitoring tools, HPCToolkit [ABF+10] aims to avoid program instrumentation and uses statistical sampling to generate the histogram of program contexts. This information is used for performance analysis within processor nodes. To capture inter-process activities, HPCToolkit intercepts process control routines such as process creation and destruction, signal handling, dynamic loading, etc.

The Periscope framework [GO10] targets MPI-based distributed programs to provide performance analysis at both the local level of each machine and the global level of the whole system. It is similar to Paradyn that the performance analysis is done online while the program is running. Unlike Paradyn, Periscope deploys distributed analysis agents to aggregate raw information collected by instrumentation and then integrate into performance metrics.

3.2 Taxonomy of Scheduling Methods

In this section, we discuss primary types of general scheduling algorithms. Given a set of tasks and a set of processing elements (PEs), the objective of a tradition scheduling algorithm is to produce a schedule which is defined as a mapping of tasks onto PEs, and the execution order of the tasks. The generated schedule must meet the task dependence constraints and the resource constraints, and at the same time optimise the performance.

Good surveys of different types of scheduling algorithms can be found in [SKH95] and [FRS+97].

A scheduling algorithm can also be classified as `offline` or `online`. A scheduling algorithm is offline if all the scheduling decisions are made before the execution of the system.

3.2.1 Offline Scheduling

In offline scheduling, the schedule is generated for the entire task set during compile time. To make scheduling decisions, the algorithm requires the complete knowledge of the system, for example the release time and the processing time of each task. This type of scheduling algorithms is therefore suitable for static systems where the behaviours are known before the execution. One example in this category is the work of Stone et al. that describes a method for optimal assignment on a two-processor system based on the Max Flow/Min Cut algorithm [Sto77]. Another example is the work in [Lo88] which proposes a heuristic method based on the Stone's Max Flow/Min Cut algorithm

to minimise the overall execution time and the communication delays on heterogeneous systems.

3.2.2 Online Scheduling

In online scheduling algorithms, decisions are made at runtime on the set of active tasks. Online scheduling algorithms are based on the distribution of tasks among PEs during runtime. There are three basic techniques for the task distribution of online scheduling algorithm: `centralised`, `distributed`, and `centralised mediation` [SS92a].

3.2.2.1 Centralised Scheduling

In the first technique, centralised scheduling, the task distribution is based on a central agent which collects the information of the system state. In common designs, tasks are not assigned statically to any PE but stored in in a central task pool. Based on the current state of the system, the central agent makes decision of distributing tasks among PEs. The central agent can be a physical PE, which makes the decision of task distribution based on the system state. One example of this approach is the Condor scheduler [LLM88]. The central agent can also be a global information directory of the system state which is shared and accessed by all the PEs. An example of this approach is the work in [DO87].

3.2.2.2 Distributed Scheduling

In the second technique, distributed scheduling, each PE has its own local task queue, and the task distribution is initiated only on some conditions, e.g. heavy unbalanced load. The task distribution can be initiated by either the sender or the receiver.

In the former case where the sender initiates the task distribution, the technique is called `work sharing`. This scheduling scheme is also known as `task migration`. In work sharing, whenever a PE generates new tasks, it attempts to migrate some of them to other PEs. Work sharing schedulers usually employ four components: information policy to specify the required information about the system state, transfer policy to determine if a PE should participate in a task migration, location policy to identify the suitable destination PE for the task migration, and a selection policy to decide which tasks are eligible to migrate. Some examples of the work sharing scheme include [BF81], [LSK97] and [LFM04].

In the later case where the receiver initiates the task distribution, the technique is also known as `work stealing`. In this scheme, when a PE has no task in its local queue, it will initiate the task distribution. This PE, called **thief**, will choose a PE, called **victim**, and steal a task from the victim. This helps to maximise the utilisation of computing resources. A work stealing scheme requires a policy to determine the victim and the task to be stolen. Two of the very first ideas for the work stealing approach were described in [BS81] and [Hal84]. These two ideas have been widely used in several scheduling libraries, for example Cilk [BJK+95], TBB [RVK08], etc. Since then, several work stealing schemes have been proposed aiming at different models. Some examples include [BL99] focusing on multithreaded computations with dependencies, [TDG+11] targeting at parallel loops on shared cache multicores, and [AHL07] aiming at fork-joined multithread jobs.

3.2.2.3 Centralised Mediation

The third technique for online scheduling is centralised mediation that uses both aspects of centralised and distributed techniques. Similar to distributed scheduling, each PE in this scheme has its own task queue. Similar to centralised scheduling, one PE is chosen to be the centralised mediator which is responsible for the task distribution. When a PE is overloaded, i.e. it has too many tasks, it sends a task to the centralised mediator. Also when a PE is underloaded, i.e. it has no task, it sends a request to the centralised mediator. The centralised mediator then sends a task to the requested PE. An example of this scheduling scheme can be found in [SS92b].

In this section, we have provided an overview of different types of scheduling algorithms. Typically, there are two classes of scheduling algorithms: online and offline. As described in Section 2.4, we target in this thesis general RSPs with variable node behaviour and dynamic program structures, offline scheduling methods are not applicable. The class of online scheduling algorithms is categorised into two subclasses: centralised and distributed. Our work in this thesis introduces different scheduling strategies which fall into the categories of centralised scheduling and offline scheduling. In particular, Chapter 7 will describe centralised schedulers for RSPs on shared memory platforms. Also, Chapter 8 will propose offline scheduling algorithms to map RSPs onto distributed systems.

3.3 Scheduling Reactive Stream Programs on Many-Core Systems

3.3.1 Static Scheduling

Among data-flow models, Synchronous Data Flow (SDF) [LM87a] has been the most attractive direction for being synchronous as the name suggests. SDF differs from the conventional data-flow model in that its nodes have static input and output rates. That means the amount of messages that each node consumes and produces during its invocation is static and predefined. Generally, in an SDF program streams are uni-directional; node computations are deterministic; node communications are synchronous; and the program structure is static. These properties are used to generate the periodic schedule at compile time [LM87b]. The execution of an SDF program is simply an iteration of this schedule where the data required for the next iteration is generated in the previous iterations. A periodic schedule is usually represented by a vector of positive numbers, each of which corresponds to a node. This is the number of node invocations during the periodic schedule. For example, a periodic schedule $\{3A, 2B, C\}$ describes the execution of the SDF program as a repetition of three times invoking node A, two times invoking node B and one time invoking node C.

The problem of scheduling an SDF program now becomes the problem of scheduling its periodic schedule. Taking advantage of the periodic schedule, several strategies have been proposed to map SDF programs into many-core systems. One of them is the work in [GTK+02] which maps StreamIt, a programming language following the SDF model, to the Raw architecture [TKM+02]. In StreamIt, each node is called *filter* and the periodic schedule is called *steady state*. This work uses profiling to estimate the required computation of each filter within a steady state. Based on this information, a greedy heuristic partitioner is designed to assign filters into each cell of the Raw microprocessor. The partitioner uses fission operations to split filters with high computational requirements; and fusion operations to merge filters with low computational requirements until the number of filters is equal to the number of cells. The scheduler in this work also includes a static communication pattern between cells which can be obtained by simulation. This communication pattern is used to determine the maximum buffer size which avoids deadlock. The later work from the same authors proposes a new approach where all communications are wrapped into a single stage which is placed by the end of the steady state [GTA06]. This approach uses another greedy heuristic partitioner which scans filters in order of decreasing required computations, and assigns each of them to the processor with the least amount of computation. The bottleneck partition is the processor with the highest amount of computational load. To minimise the communication

stage, a selective fusion pass is used to fuse two adjacent filters, and the partitioner is used to remap the new program. This pass is repeated until the new bottleneck increases by more than a given threshold.

As the first applying modulo scheduling [Rau94] to SDF, the authors of [KM08] aim to maximise the concurrent execution of nodes and at the same time to overlap the communication and computation activities. The main idea is to construct a software pipeline based on the steady state, i.e. start executing a producer node and its consumer node at different stages. This excludes intra-stage synchronisations within a steady state. The only synchronisation occurs by the end of each stage to guarantee that all inputs required for performing a stage have been generated by the previous stage. The work in [KM08] is part of a StreamIt compiler for the Cell architecture. The first step in the proposed algorithm is to split nodes by using fission operations and to partition the StreamIt program. The splitting and partitioning problem is formulated as an integer linear program (ILP) and is solved by PERPLEX [ILO14]. The second step is to assign each node invocation in the periodic schedule to a pipeline stage for execution in such a way that all communications are overlapped with computations on processors. Note that ILP is applicable for StreamIt specifically and SDF generally because of the constraints of the static input and output rates.

Udpa et al. [UGT09a] successfully used ILP and modulo scheduling to compile StreamIt on GPUs. This work also introduces a buffer mapping scheme to exploit the high memory bandwidth in GPUs. Their later work extends the approach for hybrid architectures composed of CPUs and GPUs [UGT09b]. The authors also propose a heuristic partitioning algorithm to divide the StreamIt program into two set of nodes, one for CPUs and one for GPUs. The algorithm starts with some constrained nodes assigned to CPUs and the rest assigned to GPUs. Nodes from GPUs are then considered to move to CPUs depending on their speedup benefit. The METIS partitioner [KKK98] is then used to divide node invocations across the CPUs and GPUs.

To optimise the performance of StreamIt programs on hybrid architectures, the authors of [dOCLB10] introduced restructuring methods to remove redundant synchronisation before using an existing partitioner, e.g. METIS [KKK98], to partition the StreamIt program between CPUs and a GPU. To derive the local schedule for each CPU and the GPU, the authors use the algorithm proposed in [KM08] to generate two nested modulo schedules. The outer one is for the GPU which works on large buffer sizes and the inner one is for CPUs with small buffer sizes.

Since ILP solvers require exponential solving time in the worst case, it is inefficient to recompile SDF programs whenever the availability of resources (e.g. CPUs, memory) changes. To address this problem, *Flextream* is proposed as an adaptive scheduling

algorithm combining static scheduling and dynamic adaptation [HCK+09]. For the static scheduling, Flextream first performs a pass where SDF nodes are replicated to generate the relevant amount of parallelism. With the assumption that all n PEs are available, Flextream uses ILP to generate modulo schedule including n work assignments and their local stage assignments. Before executing the SDF program, Flextream will carry out the adaptation phase depending on the availability of the CPUs. With m unavailable CPUs, Flextream chooses m work assignments with the least number of nodes and schedules them on the $(n - m)$ available CPUs. The adaptive schedule uses heuristic strategies to distribute those nodes to the CPUs, adjust stage assignment within each CPU, and fit the buffer allocation with the memory availability.

To avoid using ILP solvers, the work of [FKBS11] introduces an approximation strategy to maximise the throughput of SDF programs on multi-core systems. When an SDF program has only one entry node, the throughput is equal to the arrival rate, and also the input rate of the entry node. Based on the single entry node constraint, this work derives data rate functions of filters depending on the arrival rate. Quantitative analysis is then used to identify bottlenecks and transform the SDF programs to remove bottlenecks. Finally, a 2-approximation algorithm is employed to map each node of the SDF programs to CPUs so that the arrival rate is maximised.

Although modulo scheduling is helpful for increasing the concurrent execution, it has some disadvantages: i) requiring barrier synchronisation; ii) supporting only fixed input and output rates; and iii) not sufficiently managing feedback loops. The work of [PD10] has aimed to tackle these problems. This work includes two phases: team formation and atomisation. The first phase groups nodes on the same core into teams by using a greedy heuristic to maximise the gain as the ratio of synchronisation reduction to the buffer size growth. Unlike modulo scheduling, this approach requires only steady state within each team instead of steady state of the entire SDF program. Team communications are implemented by a blocking mechanism, i.e. a node gets blocked when reading an empty stream or writing to a full stream. To generate a static schedule of nodes within a team, existing methods like LISF [BBHL02] are used. The authors also suggest to partition SDF programs before applying this scheduling phase, although it is unclear how the partitioning is performed. The second phase is to amortise the inter-team communication overhead by finding the trade-off between the synchronisation cost and the buffer size.

A recent approach is to use machine learning to partitioning StreamIt programs for multi-core systems [WO10]. The approach first uses supervised machine learning to predict the ideal structure of the partitioned program and then selects from all partition possibilities the nearest one to the ideal structure.

By taking advantage of static properties of SDF programs, all the above approaches have been proposed for statically scheduling SDF programs on many-core systems. The work on this thesis however tries to address a more general class of stream program which allows variable node behaviours, asynchronous inter-node communication and dynamic program structures. Therefore, the static scheduling strategies proposed for SDF are inapplicable for general RSPs.

3.3.2 Dynamic Scheduling

Static scheduling has been shown to be efficient in SDF and StreamIt for their static input/output rate and static program structures. However, these also restrict the class of applications that they can support. For this reason, the work in [ZLRA08] introduces a Multi-core lightweight Streaming Layer (MSL) that supports both predictable and unpredictable stream programs on the Cell architecture. Although it is not clearly defined, predictable stream programs can be understood as SDF or StreamIt programs with static input/output rates and static structures. MSL uses the strategy of static scheduling in [GTK+02] to target this type of stream program. A stream program is unpredictable when it is difficult to estimate the execution time of each node invocation, or when the execution time of each node varies during runtime. For these cases, it is difficult to derive a static schedule with good load balance. The authors in [ZLRA08] propose to use a central-based scheduler that maintains the state of the program and tries to optimise the throughput by load balancing. The proposed scheduler does not statically map nodes to CPUs, but instead dynamically sends nodes to be executed on CPUs for a number of iterations. The selection policy has not yet been investigated and is stated to be based on the amount of input and output data of each node.

The approach in [CC09] proposes to use the notion of back-pressure on streams to obtain load balance. The approach first uses existing techniques to statically map nodes, which are called *filters* in this work, onto multi-core platforms. Then profiling is used to identify bottleneck filters which are stateless, i.e. filters provide the same output for the same input regardless of the previous sequence of input. For being stateless, multiple instances of these filters can be created and connected in parallel without changing the program's semantics. This approach provides mechanisms to preserve the message order. Copied instances of these filters are mapped onto the platform by using ILP-based analysis to maximise their individual throughputs. The execution of those copied instances are activated during runtime based on the back-pressure on the current capacity of their output streams.

Similar to the approach in [ZLRA08], the work in [ME13] proposes a central-based scheduler facilitated with dynamic load balancing. In this approach, scheduling decisions are based on the fill level of the streams. Thanks to its focus on stateless nodes, the approach also allows concurrent execution of each node to increase the data parallelism while preserving the message order. When a CPU finishes its execution on a node, the scheduler decides on which node it should execute next. If the input stream of the node is empty, the producer of the current node is chosen. If the output stream of the node is full, one consumer of the current node is executed next. Otherwise, the CPU continues working on the current node. When the parallelism level of the current node exceeds its threshold, the CPU will choose a new node based on a probabilistic scheme.

Supporting S-Net programs which is more general than SDF with no constraints on node behaviour nor on the program structures, LPEL uses dynamic mapping and dynamic scheduling [Pro10]. An S-Net program is composed of boxes as computational nodes; and combinators are used to connect these boxes. Before being scheduled by LPEL, boxes and combinators are wrapped as box tasks and non-box tasks respectively. With the reasoning that box tasks usually include heavy computation while non-box tasks requires very little computation, LPEL uses two different round-robin mappers: one for box tasks and one for non-box tasks. LPEL also uses a round-robin scheduler for local task scheduling within each core.

Another approach to scheduling S-Net programs is introduced in [GG13] where load balancing is obtained by work stealing. In this model, a CPU can execute a node if it holds at least one *reading license* for every input stream of the node. A reading licence is a logical representation for a message on the stream and does not indicate any specific message. If a CPU holds n reading licences of a stream, that means it can read n messages from the stream, and it can choose to read each of n messages at any time. The model is designed so that the number of messages on a stream is always equal to the total number of reading licences on that stream. Each CPU tries to execute nodes toward the exit if they can. When a CPU can not proceed, it steals from other CPUs reading licences on the stream towards the entry.

Although the work in [TE92] targets multihop radio networks, we include it here as they are structurally similar to RSPs. In that work, the authors propose a dynamic scheduler where the node priority is defined by the queue length of its input and output streams. With the constraint of static network structure, the authors are able to prove that their scheduling leads to maximal throughput for any input arrival rate where a stable schedule (bounded message queues) is possible.

Although static properties of SDF help to ease the burden on scheduling, they limit the range of applications. Therefore, some dynamic scheduling approaches such as the

ones in [ZLRA08] and [ME13] attempt to address general stream programs. However the work in [ZLRA08] only presents some general requirements for the selection policy but does not propose any concrete design. The ideas in [ME13] have some similarities with our work in Chapter 7 in the way the fill level of streams is used in the selection policy. These are different from ours in that they require the RTS to determine a node to be stateless. In addition, they focus on load balancing without any reasoning about throughput and latency. The LPEL scheduler described in [Pro10] uses simple round-robin strategies and does not provide good performance as shown in Chapter 7. Similar to our approach in Chapter 7, the work of Tassiulas et al. [TE92] has a similar way of defining task priority based on the queue length of input and output streams. Focusing on maximising the throughput, the authors are able to prove that their scheduling guarantees maximal throughput. However, this is only applicable for stream programs with fixed structures. The approach in [GG13] uses the same heuristic like ours that the task priority is higher when it is closer to the exit. Unlike ours, the approach does not use a central-based scheduler but instead uses work stealing to obtain load balance. In addition, by aiming for minimal total execution time the approach is rather applicable for non-reactive system where the sequence of inputs is finite and short.

3.4 Scheduling Stream Programs on Distributed Systems

Unlike many-core systems, PEs on a distributed system do not share memory and the communication cost among them is significant. Dynamic scheduling therefore seems unfeasible. To schedule stream programs on a distributed system, most of the approaches aim to find the best mapping where each node of the stream program is statically assigned to a PE of the distributed system. All of this work requires knowledge of the average data rate on each stream and the average load of each node. These values can be obtained by profiling or derived statically in the case of SDF.

One of the first attempts to map SDF programs onto homogeneous multiprocessor architecture is the work of Sih and Lee [SL93]. This approach includes four stages and aims to divide the SDF program into a set of partitions, each of which is allocated to a PE. The goal of this approach is to minimise the *make-span* which is the execution time of a single periodic schedule and also is the inverse of the throughput. In the first stage, a technique called *declustering* is employed to analyse the trade-off between parallelism benefits and inter-PE communication costs. The result of this analysis is the set of streams likely to be the connectors between the final partitions. Temporarily removing these streams forms groups of nodes. Each group is called a basic cluster. In the second stage, these clusters are repeatedly combined in a pairwise fashion to create a binary

tree whose leaves are basic clusters. The cluster combinations are formed by considering the inter-cluster communications and the parallelism relationships. Examining this binary tree, the third stage starts with the top level and maps it to the first PE. It then traverses the binary tree from the top level to the bottom one. At each level, each cluster is decomposed into its sub-clusters which are also its child-nodes on the binary tree. One of the sub-clusters is chosen to move to another PE so that the *make-span* is minimised. The fourth stage considers breaking out even the basic components if it can achieve better load balancing.

The approach in [CRA09] is designed to map a class of stream programs, which are variants of Kahn process networks with some SDF properties, onto heterogeneous multiprocessor systems. The approach uses a cost function which is defined as the maximum of the computational cost of each PE; and the communication cost between each pair of PEs. This cost function turns out to be inversely proportional to the throughput. The work proposes a 2-phase partitioning algorithm to minimise the cost function. The first phase is to recursively bi-partition both the stream program and the platform of PEs. Partitioning the stream program aims to minimise the cost function. Partitioning the platform aims to maximise a function that balances two objectives. These equalise the CPU capacity on each partition, and at the same time minimise the total interconnection among partitions. The second phase, refinement, tries to get rid of bottlenecks lying on the computation of PEs. This phase also considers some other constraints of convexity. The authors claim that the convexity is a guideline to avoid long pipelines and therefore to reduce the memory requirement as well as the latency. However, no proof has been provided and also the experiment focuses on only the total execution time without considering the throughput and latency.

Another approach is to use graph partitioning to map SPADE stream programs [GAW+08] into processing elements (PEs) in SYSTEM S [KHP+09]. Note that these PEs are not physical computational resources but are run-time software units. The approach includes three components. The first component, *PE Scheduler*, tries to assign PE to physical computation resources by using the longest processing time first scheme [Pin08]. The second one, *Oracle*, returns the largest PE with more than one node. This PE is passed to the third component, *Graph Partitioner* to be split into two non-empty PEs so that the sparsest cut is minimised. The *Graph Partitioner* is implemented by using the approximation algorithm in [LR99]. Based on these three main components, the authors develop two mapper variants. The first one is called *Basic COLA* that produces a *feasible* mapping where the computation requirement of each PE fits the CPU capacities of the available physical computational resources; and at the same time the total communication among PEs is minimised. This mapper variant is implemented as a repetition of three components Oracle, Graph Partitioner and PE Scheduler until a feasible schedule

is generated. The second one, *Advanced COLA*, generates a feasible mapping that meets the set of user-specified requirements and has an optimised load balance. This mapper consists of multiple phases, each of which tries to modify the current mapping to satisfy one condition. The modification is implemented by employing three components Oracle, Graph Partitioner and PE Scheduler.

As the first approach using ILP to partition the periodic schedule of SDF programs on heterogeneous architectures, the work in [MG12] aims to optimise the throughput. This work provides an ILP formulation based on the resource constraints, the scheduling constraints and the dependency constraints whilst at the same time taking advantage of stateless nodes at the granularity optimisation. The main goal of this work is to partition the periodic schedule not at the node level but at the node invocation level, i.e. a node can be executed by multiple PEs within a periodic schedule. The optimising target is the *make-span* which is inversely proportional to the throughput. One drawback of this approach is that the ILP formulation does not model well simultaneous multi-threads. This leads to the limitation that the execution on each PE is sequential. This drawback does not occur in the later work from the same authors. In this work, the authors use the Uppaal model checker [ABB+01] to solve the problem instead of using ILP [MG13].

Among these above approaches, those that aim for scheduling SDF on distributed systems do not fit in the class of stream programs that this thesis focuses on. The approach in [KHP+09] is similar to our work in Chapter 8 in that it also uses a graph partition method to divide SPADE stream programs. However, unlike our work that targets general heterogeneous distributed architectures, this approach is designed for SYSTEM S with their own properties. Also the graph partitioning strategy used in [KHP+09] tries to optimise the total communication cost which does not reflect the throughput as shown in Chapter 8. The work in [CRA09] uses similar throughput reasoning to our work and recognises the role of individual communication cost between each pair of partitions. However, when trying to remove bottlenecks, this work investigates in only those lying on the partition's computation. Without any proof, this work considers the convexity of the stream graph as a guideline to reduce the memory requirement as well as the latency.

3.5 Graph Partitioning

As a stream program is composed of a set of nodes connected by streams, it can be described as a graph, call stream graph. The problem of mapping a stream program onto distributed platforms of PEs is similar to the problem of diving the stream graph's vertices into subsets that meet some requirement. This problem is known as graph

partitioning. Graph partitioning is a classical NP-hard problem [GJS76]. There has been an enormous amount of work in this area. We limit ourselves here to only some representative work.

Graph partitioning is commonly used in various applications such as VLSI design [SK72], image processing [SM00], distributing workloads for parallel computations [Cha98], etc. The classic requirement of graph partitioning consists of two criteria: balancing vertex weight between subsets (called balance criterion), and minimising the total edge cut among these subsets (called total cut criterion).

In graph partitioning, algorithms using iterative improvement are the most common. Such an approach favours the balance criterion, i.e. divide the graph into subsets so that their weights of vertices are approximately equal and then apply refinement methods to move vertices between them to find the optimal total cut criterion. As a local search, the iterative improvement starts with an initial solution and repeatedly performs local perturbation of the current solution. For the local perturbation, iterative improvement can employ greedy heuristics such as Kernighan-Lin (KL) [KL70] and its algorithmic improvement Fiduccia-Mattheyses (FM) [FM82]. It can also use hill-climbing techniques such as simulated annealing [KGV83].

As KL/FM was shown empirically to be efficient [Pot97, JAMS91], many of its variations have been used for different purposes such as partitioning VLSI networks [Kri84], direct k-way partitioning [Trä06], etc. In addition, KL/FM is usually used in local refinements in recent multilevel schemes such as the work in [HL95a], METIS [KK98], JOSTLE [WC00], ParToH [cA99], KaFFPaE [SS12], SCOTCH [PR96]. These approaches first coarsen vertices according to some matching criterion to create a smaller graph at a new level. The coarsening stage is repeated until reaching the lowest level. An initial partitioning phase is used to generate partitions. The uncoarsening phase walks up each level, and applies local refinement based on KL/FM method. Employing the similar basic idea of multilevel partitioning, PARTY [MS04] instead uses another heuristic called Helpful-Set which is derived by theoretical analysis. KaFFPaE also employs some other local refinement techniques such as Max-Flow Min-Cut and genetic Algorithms.

Another approach, called spectral partitioning, optimises the total edge cut by using the eigenvalues and eigenvectors of the graph. Unlike the iterative improvement, this approach aims to find the global optimal point. Examples of this approach can be found in [AKY99, PSL90]. This approach is known to find good solutions but is very slow to run compared to iterative improvement.

To improve the spectral approach, there are some proposals to combine the balance and total cut criteria into a single metric, for example Ratio Cut [HK06] and Sparest Cut [ARV09, KRV06].

While most of the approaches focus on the balance and total cut criteria, they are inefficient in some specialised domains. For example, Aspect Ratio [DPSW98] and the Partition Shape [DPSW00] are shown to be a better metric for partitioning solvers using the finite element method. Most of the partitioning strategies working on these criteria use iterative improvement. Starting with a set of seeds, a growing method is used to generate corresponding subsets. The centres of those subsets are used as seeds for the next iteration. The growing method can be based on a greedy breadth-first search [DPSW00], or based on the diffusive process [MMS09].

Shown in Chapter 5, the throughput of RSPs on distributed platforms does not depend on the total cut but depend on individual cuts between each pair of partitions. Traditional partitioning algorithms are therefore not applicable for the domain of this problem.

3.6 Chapter Summary

In this chapter, we have presented a range of work related to three aspects: monitoring parallel programs, scheduling stream programs, and graph partitioning.

To the best of our knowledge, there has been no specialised monitoring work for stream programs. While there are several monitoring frameworks for parallel programs, these do not capture the stream properties required to calculate the performance of RSPs and to support scheduling of RSPs. Therefore we will propose in Chapter 6 a novel monitoring framework aiming to capture internal behaviours of RSPs. The framework provides information to calculate the performance for RSPs as well as to derive efficient scheduling strategies.

In addition, this chapter presented two basic classes of scheduling algorithms: online and offline. The class of online scheduling algorithms is then categorised into two subclasses: centralised and distributed. Different scheduling strategies proposed later in this thesis fall into the categories of centralised scheduling and offline scheduling. In particular, Chapter 7 will describe centralised schedulers for RSPs on shared memory platforms. Also, Chapter 8 will propose offline scheduling algorithms to map RSPs onto distributed systems.

This chapter also included scheduling approaches for stream programs both on many-core systems and distributed systems. As pointed out above, scheduling approaches for SDF programs are not applicable to the general class of stream programs which is the target of this thesis. Some other approaches aim for some special types of stream programs, for example the approach in [KHP+09] is specialised for SPADE stream programs. There are some approaches that attempt to address general stream programs; however they are either not fully evolved like [ZLRA08], or limited by some constraints like [ME13] and [TE92]. Aiming to support general RSPs, our work in Chapter 7 provides scheduling strategies for shared memory platforms.

Finally, due to the conceptual similarity between the problem of mapping stream programs onto distributed systems and the problem of graph partitioning, we included some representative algorithms for graph partitioning. While traditional graph partition algorithms aim to obtain load balance first and then minimise the total cut, the throughput of RSPs is a function that combines both load balance and individual cuts. These algorithms therefore are not applicable to optimise the throughput of RSPs. Our work in Chapter 8 introduces new graph partition algorithms to map RSPs onto distributed system so that the throughput gets optimised.

Chapter 4

Use Cases

In this chapter, we describe the use cases which we use in Chapter 5, Chapter 6, Chapter 7, Chapter 8 and Chapter 9 to experimentally evaluate the contributions of this thesis. These use cases are chosen for their common usages in the context of reactive systems. They are also implemented as stream programs using the language S-Net (see Chapter 2). Besides the functional description of these use cases we also describe the concurrency available by the concrete implementation in S-Net.

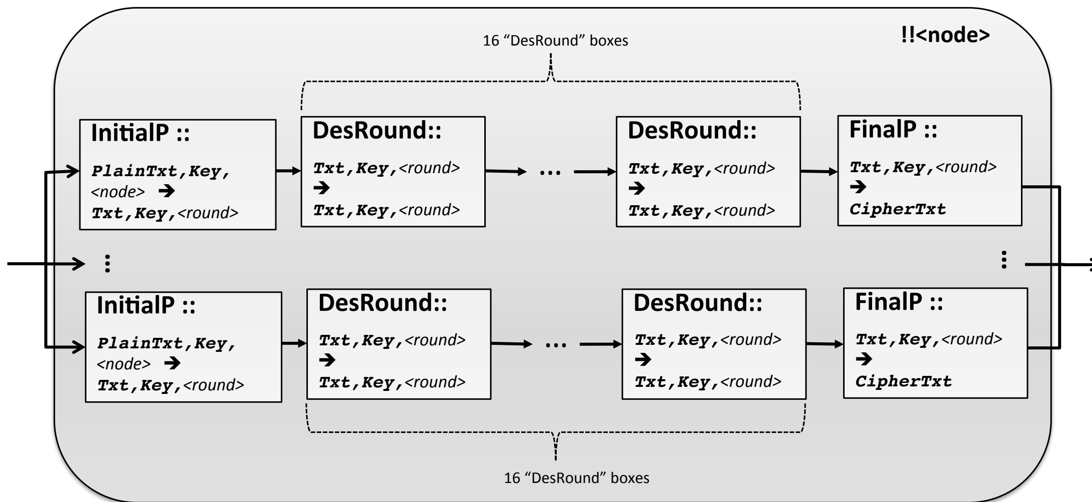
4.1 Data Encryption Standard - DES

The first use case is a DES cipher application [DES77]. This use case is denoted as **DES**. Input for this use cases is a sequence of messages, each of which contains a number of plaintext blocks and their corresponding keys. The S-Net implementation of this use case is shown in Figure 4.1. The application contains a main structure which performs the DES encryption. The main structure implements the three stages of the encryption process. The box `InitialP` applies the initial permutation and splits a block of bits into two blocks of equal size. The `RoundP` box implements the actual ciphering that is applied to the two blocks. As shown in Figure 4.1, 16 instances of this box are connected in a pipeline manner to apply 16 rounds of ciphering to the bit blocks. The box `FinalP` joins up the two blocks into one cipher text block and applies the final permutations. The execution time of each box varies depending on the size of input messages, i.e. the number of plaintext blocks within a message.

The main structure is connected by a parallel replicator. This helps to increase the level of concurrency by generating a number copies of the main structure and connecting them in parallel. The number of copies can be defined by the user or automatically assigned depending on the number of resources on the platform.

```
net Des
{
  box initP((PlainTxt, Key, <node>) -> (Txt, Key, <round>));
  box subRound((Txt, Key, <round>) -> (Txt, Key, <round>));
  box finalP((Txt, Key, <round>) -> (CipherTxt));

} connect
(
  initP .. subRound .. subRound .. subRound .. subRound .. subRound .. subRound
        .. subRound .. subRound .. subRound .. subRound .. subRound .. subRound
        .. subRound .. subRound .. subRound .. subRound .. finalP
)!!<node>;
```

(A) S-Net code



(B) S-Net structure

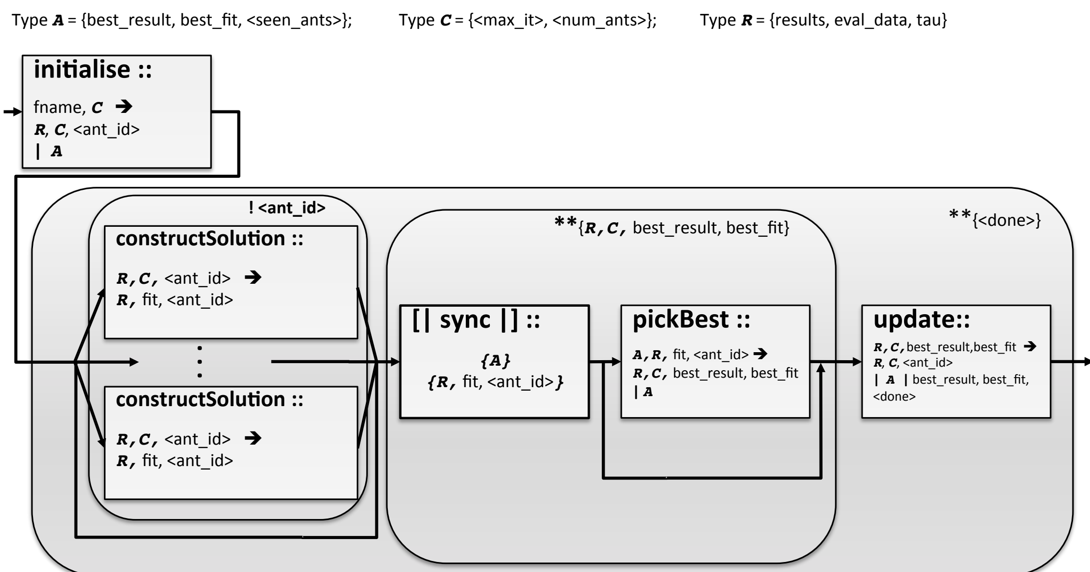
FIGURE 4.1: S-Net implementation of the DES encryption

4.2 Ant Colony Optimization - ANT

This use case implements a solver for combinatorial optimisation problems based on the behaviour of ants [DS04]. We denote this use case as **ANT**. Several ants iteratively construct solutions to a given problem and leave a pheromone trail behind. Subsequent ants use these trails as a guide and base their decisions on it, refining previously found good solutions. The S-Net implementation of the ANT use case is shown in Figure 4.2. The ants are simulated by the `constructSolution` box. The amount of parallel instances of this box determines the number of ants that are concurrently working on solutions. The subsequent stage `pickBest` analyses the solution of each ant and determines the best one, which is used to `update` the pheromone matrix that guides the ants during the next iteration. The iterative process is implemented by means of a $\star\star$-combinator that unfolds instances of the solver into a multi-staged pipeline. Concurrency may be exploited in space, by means of parallel solver instances, and in time, by overlapping the execution of multiple stages of the pipeline.

```
net ant {
  box initialise(
     (fname, <max_it>, <num_ants>)
     -> (results, eval_data, tau, <max_it>, <num_ants>, <ant_id>)
        | (best_result, best_t, <seen_ants>));

  box constructSolution(
     (results, eval_data, tau, <max_it>, <num_ant>, <ant_id>)
     -> (results, eval_data, tau, fit, <ant_id>));

  box pickBest(
     (best_result, best_fit, <seen_ants>, results, eval_data, tau, fit, <ant_id>)
     -> (results, eval_data, tau, <max_it>, <num_ants>, best_result, best_fit)
        | (best_result, best_fit, <seen_ants>));

  box update(
     (results, eval_data, tau, <max_it>, <num_ants>, best_result, best_fit)
     -> (results, eval_data, tau, <ant_id>, <num_ants>)
        | (best_result, best_fit, <seen_ants>)
        | (best_result, best_result, <done>));
}
connect initialise ..
(
  (constructSolution!<ant_id> | []) ..
   (
     [| {best_result, best_fit, <seen_ants>},
          {results, eval_data, tau, fit, <ant_id>} |]
     .. (pick_best | [] )
    ) ** {results, eval_data, tau, <max_it>, <num_ants>, best_result, best_fit}
   .. update
) ** {<done>};
```

(A) S-Net code



(B) S-Net structure

Figure 4.2: S-Net implementation of the ant colony optimisation

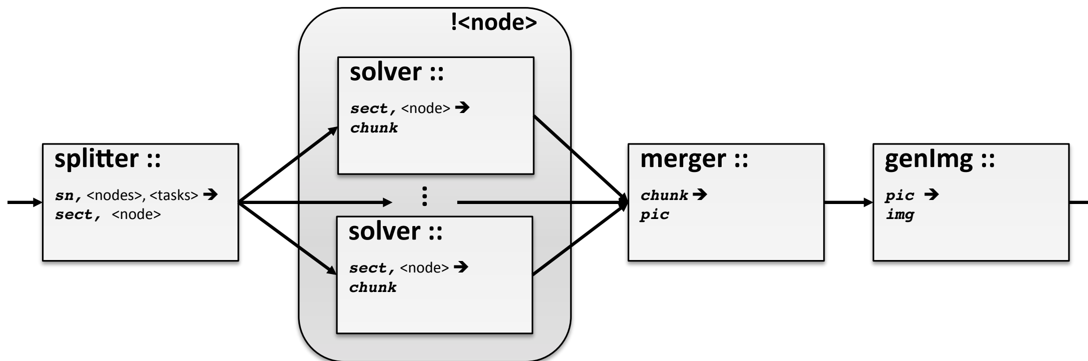
4.3 Ray Tracing - RT

Ray tracing is a technique for generating 2D images of a 3D scene by tracing the paths of light from the eye of an imaginary observer through pixels in an image plane and simulating the effects of their counters with virtual objects [Whi80]. This use case is denoted as **RT**.

Figure 4.3 shows the S-Net implementation of a distributed ray tracer. The implementation uses a fork-join pattern where the original scene that is to be rendered is broken down into several sub-scenes by the `splitter` box. Parallel `solver` instances work on the sub-scenes concurrently; the number of parallel solvers is dynamic and may be adapted through a tag parameter. The sub-scenes are collected and merged into a global result by the `merger` box before the `genImg` box transforms the computed scene into an image. The execution time of each box is dynamic depending on the size of its input messages.

```
net raytracing
{
  box splitter((sn, <nodes< <tasks>)-> (sect, <node>));
  box solver((sect, <node> -> chunk));
  box merger((chunk) -> (pic));
  box genImg((pic) -> (img)

} connect splitter .. solver!@<node> .. merge .. genImg;
```

(A) S-Net code



(B) S-Net structure

FIGURE 4.3: S-Net implementation of the ray tracing application

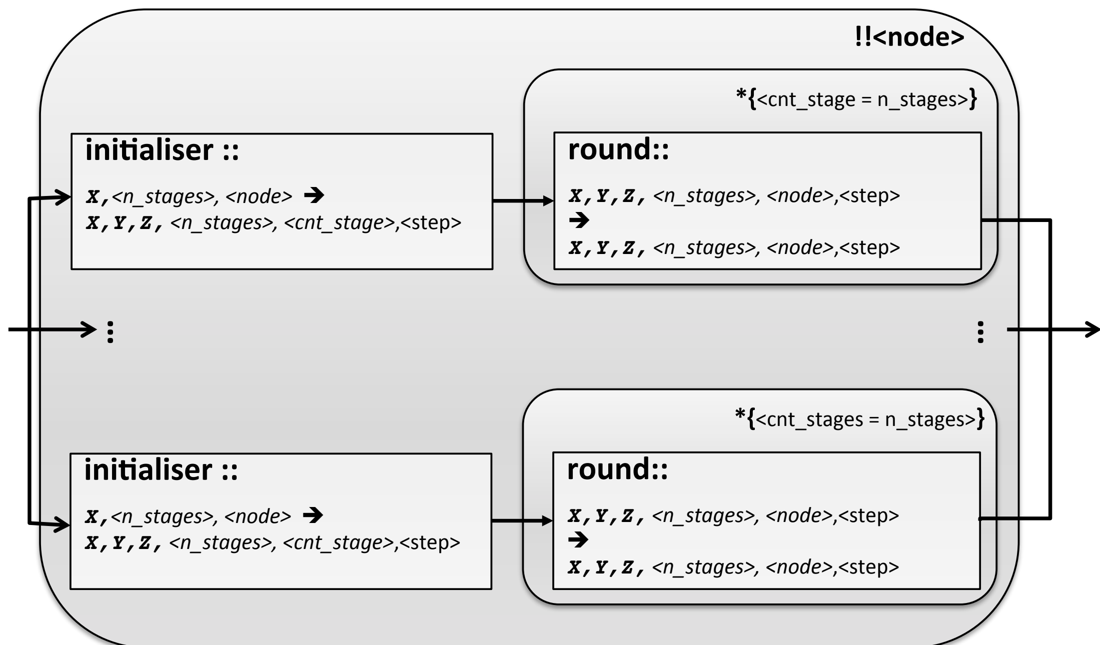
4.4 Fast Fourier Transform - FFT

Fast Fourier transform is a well known algorithm to compute the discrete Fourier transform. We denote this use case as **FFT**. The S-Net implementation of this algorithm is

shown in Figure 4.4. The application contains a main structure which performs the FFT algorithm. The main structure includes the `initialiser` box which allocates memory `Y` for to store temporary values during the transformation, generates the series of sine and cosine waves `W`, initialises `cnt_stages` as zero, and decides the step size `step` for each following FFT round. The `round` box applies `step` stages, each of which calculates the N/2 frequency spectra from N frequency spectra from the previous stage. This box also increases `cnt_stages` by `step`. The `round` box is consecutively applied by means of a $\star$-combinator until the all stages have been calculated, i.e. `cnt_stages` equals to `n_stages`. The execution time of each box varies depending on the size of the input message, i.e. the number of frequency spectra.

```
net fft
{
  box initialiser(
    (X, <n_stages>, <node>)
    -> (X, Y, W, <n_stages>, <cnt_stage>, <step>));
  box round(
    (X, Y, W, <n_stages>, <cnt_stage>, <step>)
    -> (X, Y, W, <n_stages>, <cnt_stage>, <step>));
} connect
(
  initaliser .. round*<cnt_stage = n_stages>
)!!<node>;
```

(A) S-Net code



(B) S-Net structure

Figure 4.4: S-Net implementation of the FFT algorithm

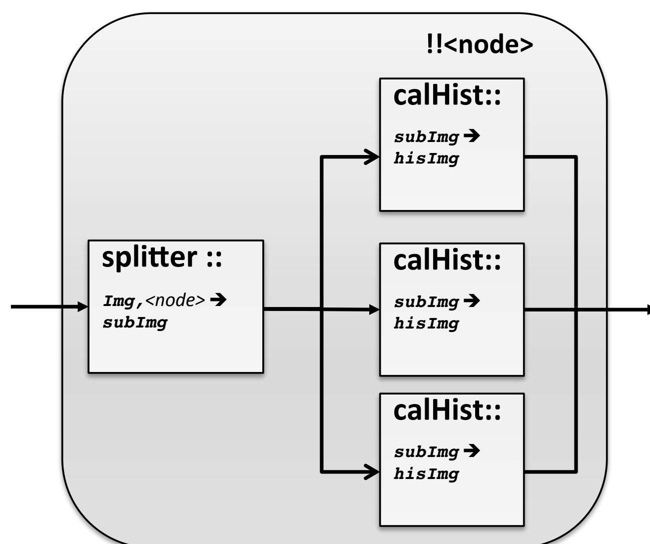
The main structure is connected by a parallel replicator. This helps to increase the level of concurrency by generating a number copies of the main structure and connecting

them in parallel. The number of copies can be defined by the user or automatically assigned depending on the number of resources on the platform.

4.5 Color Histogram Calculation - HIST

```
net histogram
{
  box split((Img, <node>) -> (subImg));
  box calHist((subImg) -> (hisImg));
} connect
(
   splitter .. (calHist | calHist | calHist);
 )!!<node>;
```

(A) S-Net code



(B) S-Net structure

FIGURE 4.5: S-Net implementation of the application of histogram calculation

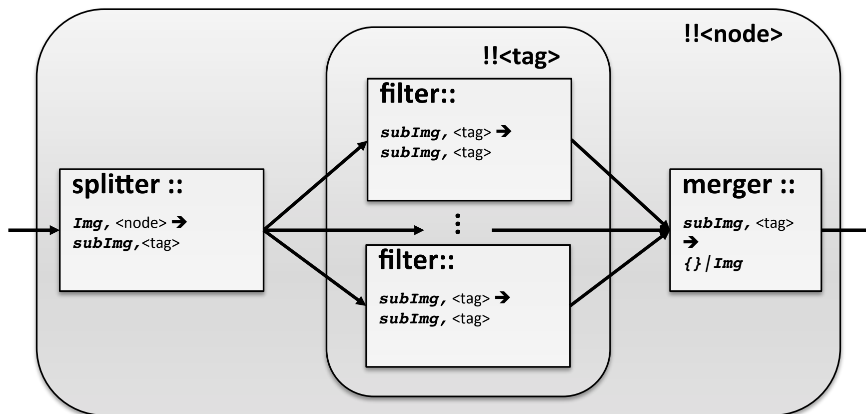
In this use case, the application receives input as digital images and calculates their RGB color histograms. This use case is denoted as **HIST**. The S-Net implementation is shown in Figure 4.5. The implementation employs a parallel replicator to generate instances of the main structure which performs the main calculation of the application. The number of instances depends on either the user's input or the number of number of resources on the platform. The execution time of each box is dynamic depending on the size of input images.

The main structure of this application starts with a `splitter` box which separates RGB channels of the image. Each of these three channels is passed to a `calHist` box to calculate the color histogram.

4.6 Image Filtering - IMF

```
net filter
{
  box splitter((Img, <node>) -> (subImg, <tag>));
  box filter((subImg, <tag>) -> (subImg, <tag>));
  box merger((subImg, <tag>) -> () | (Img));
} connect
(
  splitter .. filt!!<tag> .. merger
)!!<node>;
```

(A) S-Net code



(B) S-Net structure

FIGURE 4.6: S-Net implementation of the image filter application

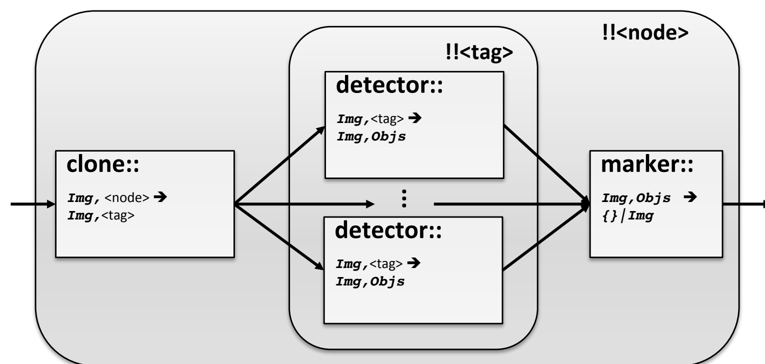
This is an application to apply different filters on image inputs. Figure 4.6 shows the S-Net implementation of this application. The idea is to split the image into several sub-images and apply the filters on each sub-image concurrently. The main structure of the application contains the `splitter` box to divide the input image into sub-images. The number of sub-images varies depending on the image size. Sub-images are assigned with different tag values so that they are passed to different instances of the `filter` box. This box applies the filters on each sub-image. Filtered sub-images are then aggregated by the `merger` box. This box returns the output image when all its sub-images have been received. The execution time of each box varies depending on the size of the input image.

To increase the level of concurrency, the implementation employs a parallel replicator to generate multiple instances of the main structure. The number of instance can be defined by the user or can be automatically derived depending on the number of resources on the platform. The application is denoted as **IMF**.

4.7 Object Detecting - OBD

```
net object_detector
{

  box clone((Img, <node>) -> (Img, <tag>));
  box detector((Img, <tag>) -> (Img, Objs));
  box marker((Img, Objs) -> () | (Img));
} connect
(
  split .. (detector!!<tag>) .. marker
)!!<node>;
```

(A) S-Net code



(B) S-Net structure

FIGURE 4.7: S-Net implementation of object detector application

The main function of this use case is to detect a set of objects on input images. The use case is denoted as **OBD**. As shown in Figure 4.7, the S-Net implementation consists of a parallel replicator to create multiple copies of the main structure which perform object detecting concurrently. Similar to the above use case, the number of copies can be defined by the user or can be automatically derived from the number of resources on the platform.

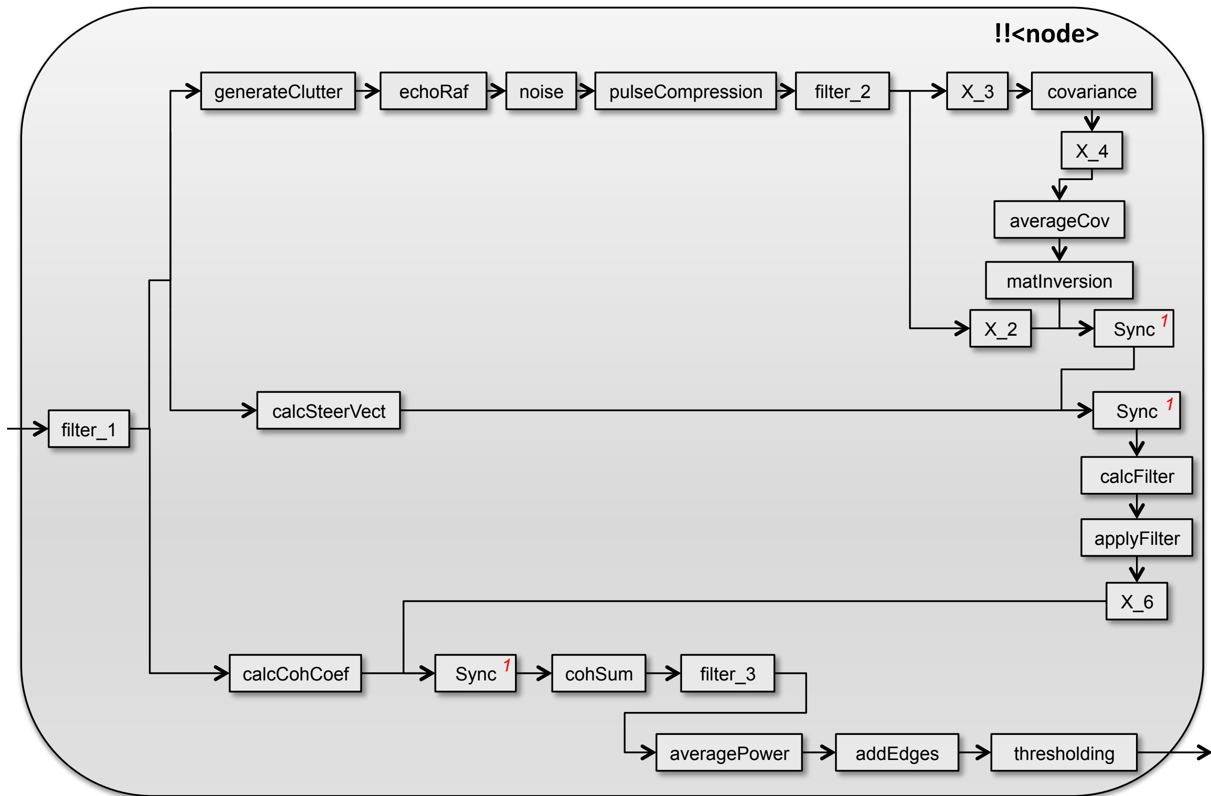
The main structure starts with the `clone` box to create multiple copies of the image. When deployed on shared memory platforms, each copy of the image is a pointer to the actual image. Each copy is coupled with a tag value to indicate the type of objects to be detected. The pair of image copy and tag value is sent to the `detector` box. Depending on the tag value, this box uses an appropriate cascading classifier to search for the desired objects in the image. The positions of detected objects are then used by the `marker` box to mark the object in the original image. This box returns the output image when all objects have been marked. The execution time of each box varies depending on the size of input images and the number of detected objects that they contain.

4.8 Moving Target Indicator - MTI

```
net mtistap
{
  box generateClutter((clutter_rnd_array_2d, <node>) -> (array_2d));
  box echoRaf((array_2d) -> ( array_3d));
  box noise((array_3d, noise_rnd_array_1_3d, noise_rnd_array_2_3d) -> ( array_3d));

  box pulseCompression(( array_3d) -> ( pulse_array_3d));
  box X_3((input_3d_1) -> (array_3d));
  box covariance((array_3d) -> (array_4d));
  box X_2((input_3d_2) -> (array_3d_signal));

  box X_4((array_4d) -> (array_4d));
  box averageCov((array_4d) -> (array_3d));
  box matInversion((array_3d) -> (inv_array_3d));

  box calcSteerVect((empty_array_3d) -> (calc_steer_array_3d));
  box calcFilter((inv_array_3d, calc_steer_array_3d) -> (array_4d_filter));
  box applyFilter((array_3d_signal,array_4d_filter) -> (array_4d_filtered));

  box X_6((array_4d_filtered) -> (array_4d_filtered));
  box calcCohCoefs((empty_array_2d) -> (coh_array_2d));
  box cohSum((array_4d_filtered, coh_array_2d) -> ( sum_array_3d));

  box averagePower((sum_3d_1) -> (array_1d));
  box addEdges((sum_3d_2, array_1d) -> ( array_3d));
  box thresholding((array_3d) -> ( threshold_array_3d));
} connect
(
  [ {clutter_rnd_array_2d, noise_rnd_array_1_3d, noise_rnd_array_2_3d,
        empty_array_3d, empty_array_2d}
    ->
    {clutter_rnd_array_2d, noise_rnd_array_1_3d, noise_rnd_array_2_3d};
    {empty_array_2d};
    {empty_array_3d} ]
   ..
   (calcCohCoefs |
      (
         (calcSteerVect |
            (generateClutter .. echoRaf .. noise .. pulseCompression
                  .. [ {pulse_array_3d} ->
                        {input_3d_1 = pulse_array_3d};
                        {input_3d_2 = pulse_array_3d} ]
                  .. (
                       X_2 |
                       (X_3 .. covariance .. X_4 .. averageCov .. matInversion)
                     )
                  .. ([| {inv_array_3d}, {array_3d_signal} |]
                       *{inv_array_3d, array_3d_signal})
            )
         )
         .. ([| {inv_array_3d,array_3d_signal},{calc_steer_array_3d} |]*
             {inv_array_3d,array_3d_signal,calc_steer_array_3d})
         .. calcFilter .. applyFilter ..X_6
      )
    )
    .. [|{array_4d_filtered},{coh_array_2d}|]*{array_4d_filtered, coh_array_2d}
    .. cohSum
    .. [ {sum_array_3d}->{sum_3d_1=sum_array_3d,sum_3d_2=sum_array_3d} ]
    .. averagePower .. addEdges .. thresholding
)!!<node>;
```

(A) S-Net code

This use case aims to detect moving objects on the ground from an aircraft. The input is a periodic sequence of echo radar pulses from the ground. The application



[1] *"Sync" is a combination of a synchro-cell inside a serial replicator to merge the outputs from two branches of the prior parallel compositor*

(B) S-Net structure

FIGURE 4.8: S-Net implementation of the moving target indicator application

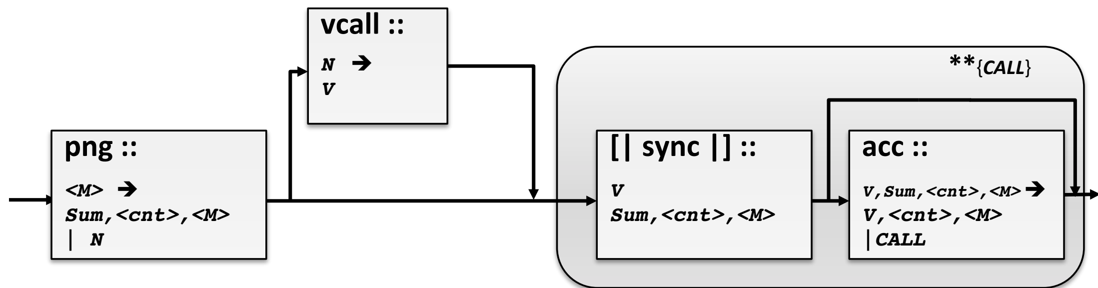
uses the method of Space Time Adaptive Processing (STAP) to process these radar signals to distinguish moving objects from others. The S-Net implementation is shown in Figure 4.8. To increase the parallelism, the application is implemented with a parallel replicator to create multiple instances of the main structure. A number of instances is either defined by the user or generated dynamically depending on the number of resources available. The detailed implementation of each box in the main structure can be found in [PHG+10].

4.9 Monte Carlo Option Price - MC

This application calculates option prices using the Monte Carlo method [BS73]. We denote this application as **MC**. The S-Net implementation of the application is shown in Figure 4.9. The `png` box generates random numbers while the `vcall` box calculates the underlying assets. The `acc` box accumulates all these underlying assets. It then

```
net OptionPrice
{
  box png((<M>) -> (SUM, <count>, <M>) | (N));
  box vcall((N) -> (V));
  box acc((V, SUM, <count>, <M>) -> (SUM, <count>, <M>) | (CALL));
} connect
  png .. (vcall | []) ..
  (
    ([|{V}, {SUM, <count>, <M>}|] .. (acc | [])
  ) **{<CALL>});
```

(A) S-Net code



(B) S-Net structure

FIGURE 4.9: S-Net implementation of the Monte Carlo option price application

calculates the average value and produces the option price by applying the discount factor.

Note that this implementation does not map well to RSPs as it has too fine-grained of concurrency, i.e. each box contains a very small amount of computation. Usually this kind of implementation is not sufficient in stream programming as the cost of stream communication surpasses the benefits of concurrency. This use case is included here as it will be used in Chapter 6 to show the monitoring overhead in the extreme case.

Chapter 5

Performance Analysis for Reactive Stream Programs

This chapter investigates the throughput and latency which are common performance metrics in the context of RSPs. The focuses include the formal definitions of these two metrics, the relationship between them and the arrival rate of input data, and their quantitative formulas on both shared-memory and distributed systems.

5.1 Performance Metrics

RSPs are similar to communication networks in the sense that they transfer messages from one end to another via interconnected nodes. For this reason, the performance of RSPs are evaluated with similar metrics to communication networks, i.e. throughput and latency. However, unlike communication networks, nodes in RSPs contain extensive computations. These computations need to be executed on a mutual platform of physical resources (e.g. CPUs). This chapter investigates throughput and latency and their relations in the context of RSPs.

5.1.1 Throughput

In similarity with communication networks, the throughput of RSPs is the rate of completely processing external input messages. Throughput is measured in messages per time unit. In contrast to communication networks, nodes in an RSP perform their computations on a shared platform of physical resources. Thus the throughput of RSPs depends highly on the scheduling policy.

5.1.2 Latency

The latency is the delay experienced in the system, i.e. the delay to transfer one message from one entry point to an exit point. In an RSP, the latency of an external input message is the time interval from when the message arrives at the program to when it is completely processed.

On a platform with shared physical resources, the latency of an external input message also depends upon the scheduling policy deciding when a node can perform its computations. Once an external input message arrives, depending on the scheduler it may or may not be processed immediately. In the latter case, the message has to wait in *input queues*. The length of input queues can grow infinitely if the external input messages arrive faster than the RSP can consume. Similarly, intermediate messages may not be processed immediately after produced. They may have to wait inside streams, for example because of physical resource limitations; or because of the scheduling policy; or because processing them requires synchronising with other messages which are not yet available.

These two waiting periods together with the computation period form the latency. As the period of waiting in input queues can be separated from two others, we propose three different types of latency as follows:

- *Queuing Latency* is the time interval an external input message waits in one of the input queues.
- *Processing Latency* is the time interval from when an external input message is consumed by the RSP until it is completely processed.
- *Overall Latency* is the sum of queuing latency and processing latency.

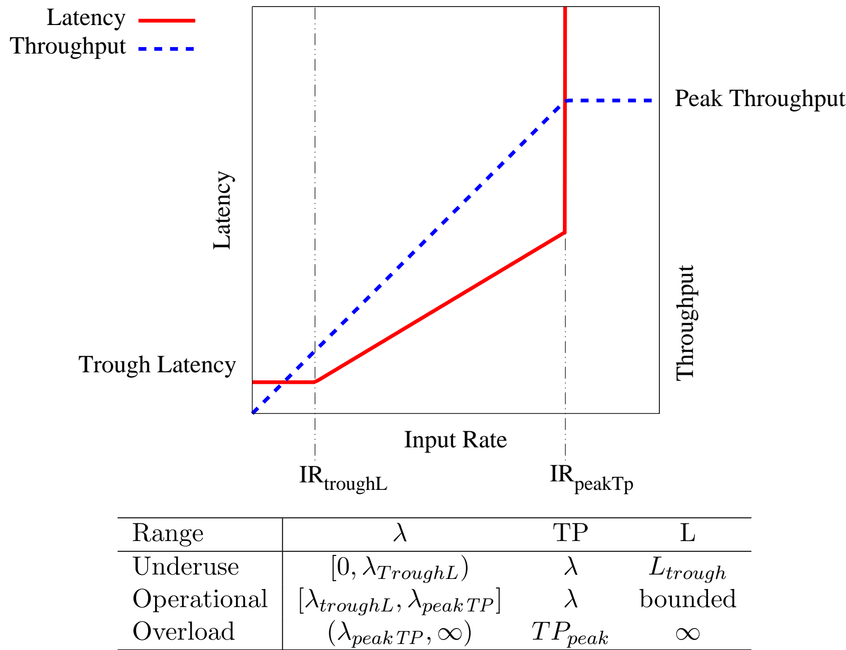
All types of latency may vary for different external input messages for different reasons. One reason is that each message may require a different amount of computation and therefore requires a different amount of processing time. For example, the Image Filter application requires more time to process a larger image. Another reason is the scheduler which decides how long a message has to wait before getting processed. For the above reasons, the *average latency* is usually used to evaluate the performance of an RSP. The average latency is calculated as the arithmetic mean of latencies of all external input messages. In the context of this thesis, latency is used to indicate the average latency unless it is explicitly stated as latency of a specific message.

5.2 Performance with Different Arrival Rate

5.2.1 Theoretical Analysis

As unidirectional streams can be considered as queues of messages which are required to pass through connected nodes, an RSP can be considered as a queuing system. Let λ be the *arrival rate* at which external input messages arrive to the RSP. This section discusses the relations between the arrival rate and the performance in queuing systems in general and in RSPs specifically.

There are three ranges of the the arrival rate with different effects on the throughput and latency. These ranges are shown in Figure 5.1 and explained in more detail as follows. Let $\overline{M_{cp}}$ be the average number of external input messages currently processed by the RSP.



Range	λ	TP	L
Underuse	$[0, \lambda_{TroughL})$	λ	L_{trough}
Operational	$[\lambda_{troughL}, \lambda_{peakTP}]$	λ	bounded
Overload	$(\lambda_{peakTP}, \infty)$	TP_{peak}	∞

FIGURE 5.1: Theoretical variation of latency and throughput in different arrival rate ranges

Underuse Range. At the time $t = 0$, when the first external input message arrives it is processed exclusively by all physical resources. The message's latency therefore reaches its smallest value. If λ is low enough so that $\overline{M_{cp}}$ is not greater than one, every external input message once arrived is processed immediately and exclusively by all physical resources. The latency is the smallest possible value for every message. The average latency is therefore smallest and this value is called **trough latency** (L_{trough}).

In this scenario, any external input message, once it has arrived, is completed after a period of L_{trough}. The throughput therefore equals the arrival rate of input messages, i.e. λ.

The highest value of λ at which $\overline{M_{cp}}$ is still not greater than one and the latency is as low as L_{trough} is called $\lambda_{TroughL}$. The range $[0, \lambda_{TroughL})$ is called the *underuse range* as there are time periods where physical resources are inactive waiting for messages.

Operational Range. When λ exceeds $\lambda_{TroughL}$, $\overline{M_{cp}}$ is greater than one and external input messages are no longer processed exclusively. Instead the processing of external input messages tends to overlap. When an external input message arrives, the physical resources are currently processing other messages. The message has to share resources with other messages or even has to wait in input queues. The latency is therefore higher than L_{trough}.

If $\overline{M_{cp}}$ is bounded, the latency L is also bounded. In this case, an external input message once arrived is completed after a time period of L. As with the underuse range, the throughput in this range also equals the arrival rate λ.

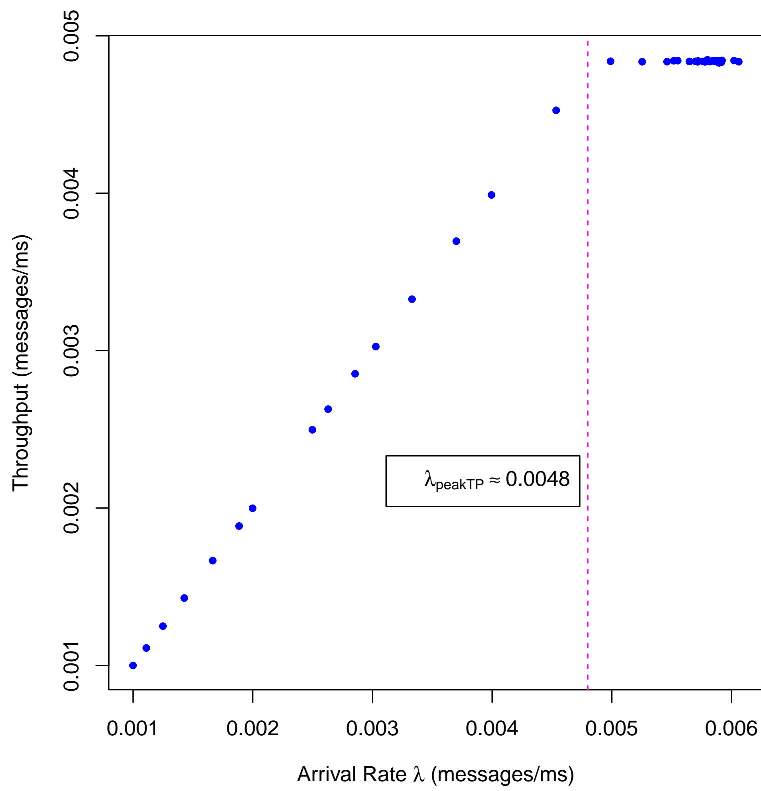
The highest value of λ at which $\overline{M_{cp}}$ is still bounded defines the upper limit of the operational range. At this value of the arrival rate, the system reaches its highest throughput called **peak throughput** (TP_{peak}). This TP_{peak} value is also the maximum arrival rate that the system can cope with. If the arrival rate exceeds this value, the system will get saturated and that causes $\overline{M_{cp}}$ to become infinite. This arrival rate is called λ_{peakTP}. We therefore have the *operational range* as $[\lambda_{TroughL}, \lambda_{peakTP}]$, and λ_{peakTP} is equivalent to TP_{peak}.

Overload Range. The last range is defined as $(\lambda_{peakTP}, \infty)$. Once λ is higher than λ_{peakTP}, $\overline{M_{cp}}$ becomes infinity. This means external input messages on average have to wait for an infinite amount of time before being processed. This makes the latency infinite.

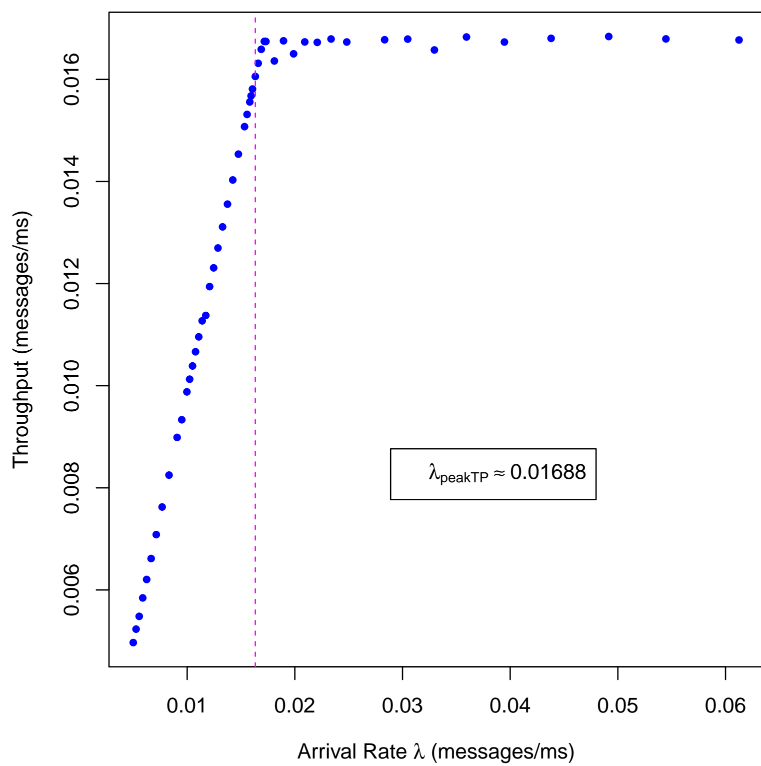
In this circumstance the system cannot consume external input messages as fast as the arrival rate. The system's throughput therefore cannot exceed TP_{peak}. As the system is saturated and cannot keep up with the requested arrival rate, this range is called the *overload range*.

5.2.2 Experimental Verification

To verify the theoretic analysis above, an experiment is carried out with the DES benchmark which applies DES encryption on messages of 8 KB. The benchmark is performed

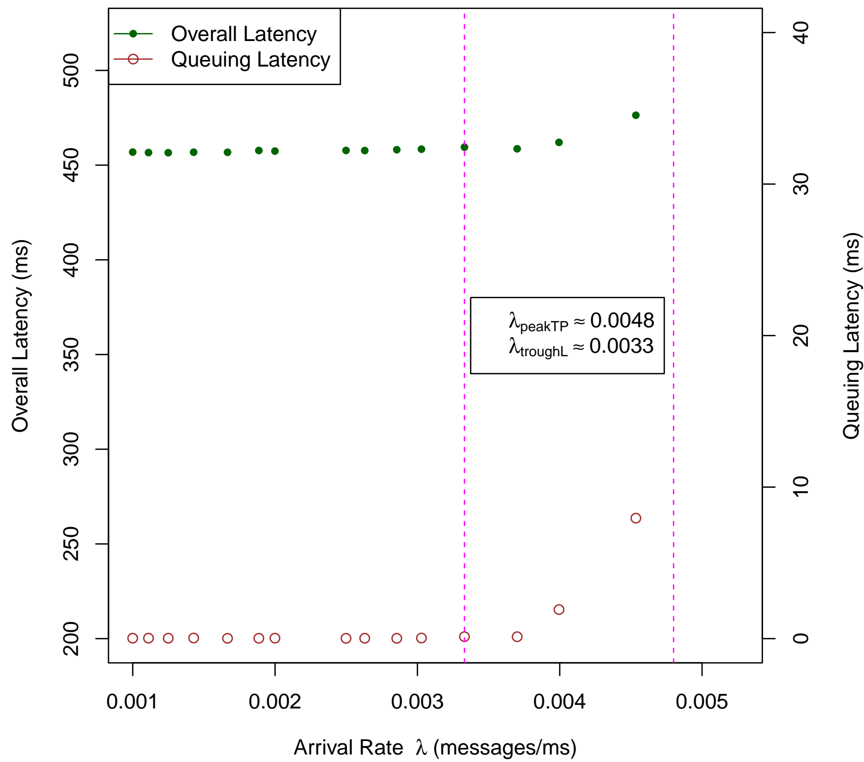


(A) DES on DS scheduler

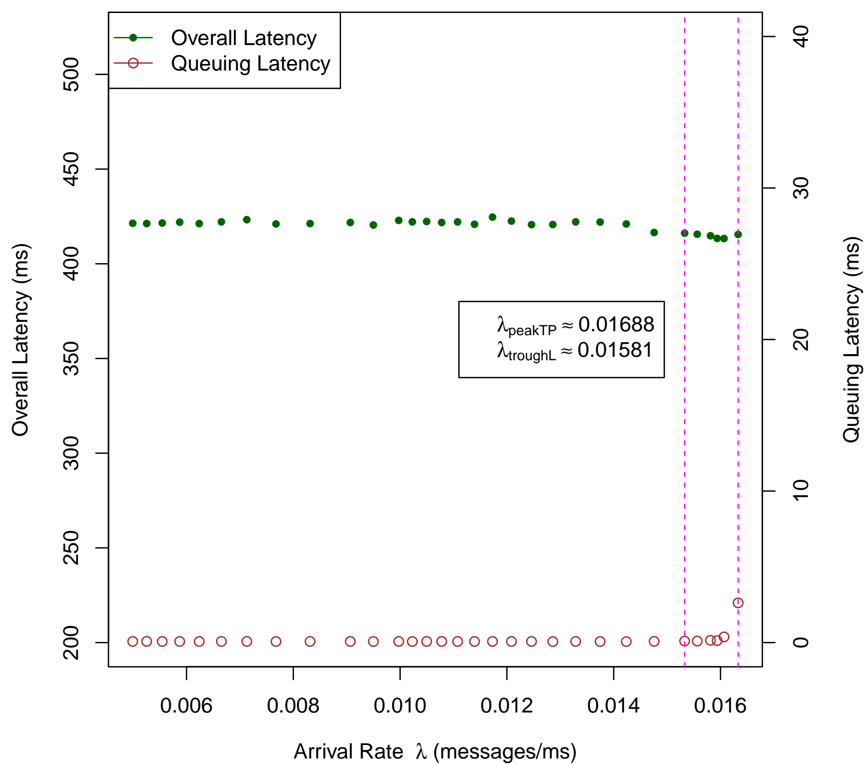


(B) DES on CS-dbp scheduler

FIGURE 5.2: Throughput within different ranges of arrival rate



(A) DES on DS scheduler



(B) DES on CS-dbp scheduler

Figure 5.3: Overall latency and queuing latency in the underuse and operational ranges

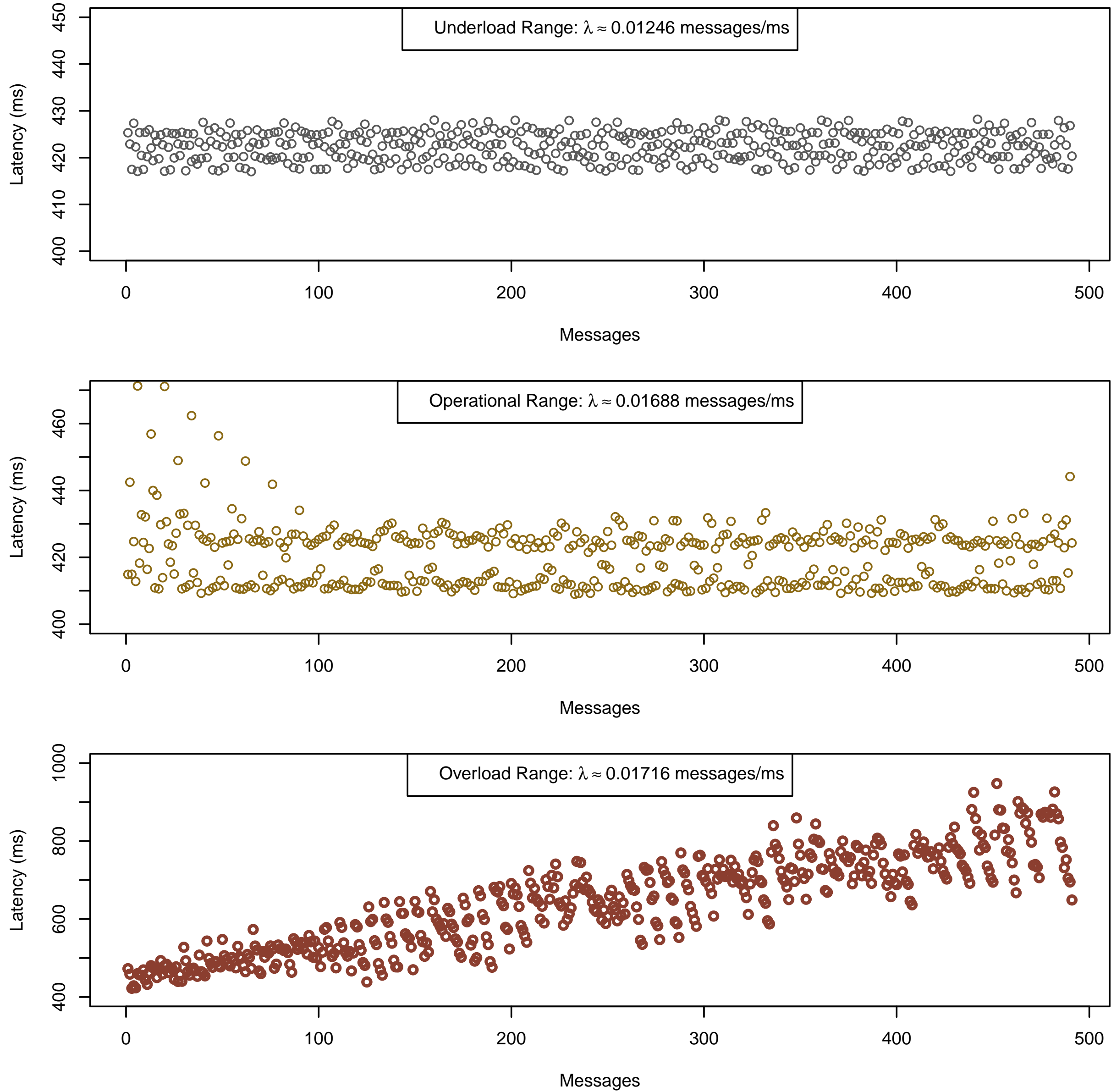


FIGURE 5.4: Overall latency of individual external input messages of DES on CS-dbp

with two different schedulers: the default scheduler of LPEL, and the centralised scheduler with demand-based priority. The former is denoted as **DS** and a detailed description is presented in Section 2.5. The later is denoted as **CS-dbp** and a detailed description is presented in Section 7.2.

Figure 5.2 is shown to verify the theoretical analysis of the throughput behaviour in different ranges of λ. The figure demonstrates the throughput of the DES application with different λ values. The figure shows that for both schedulers CS-dbp and DS, the throughput increases and is approximately equivalent to λ. When the arrival rate reaches the value of λ_{peakTP}, the throughput stops increasing and stays stable instead. This matches the throughput behaviour as shown in Figure 5.1 where the throughput

is equivalent to λ in the underuse and operational range. In the overload range where λ exceeds the λ_{peakTP}, the throughput remains unchanged. The throughput behaviour shown in Figure 5.2 therefore verifies the theoretical analysis discussed in Section 5.2.1.

Figure 5.3 shows the empirical evidence supporting the theoretical analysis of the latency behaviour in the underuse and operational ranges. The figure shows the overall latency and queueing latency of both schedulers CS-dpb and DS. The overall latency and queueing latency values are obtained by observing the first 500 external input messages of the DES application. Figure 5.3 shows that on both schedulers, the overall latency and queuing latency the overall latency and queuing latency are bounded while λ is in the underuse and operational ranges. Within the underuse range, the queuing latency is approximately zero and the overall latency remains stable until λ reaches the value of $\lambda_{TroughL}$. In the operational range $[\lambda_{TroughL}, \lambda_{peakTP}]$, the queuing latency is no longer close to zero and increases when λ increases. The overall latency also rises in this range. The behaviour of queuing and overall latency shown in Figure 5.3 verifies the theoretical analysis in Section 5.2.1.

Figure 5.4 shows the empirical evidence supporting the theoretical analysis of the latency behaviour in the overload range. According to the analysis in Section 5.2, the latency in the operational range is infinity. The latency of external input messages increases gradually and eventually reaches infinity. As it is not feasible to observe the latency of an infinite number of external input messages, we observe the trend in latency of individual external input messages. Figure 5.4 shows the latency of 500 continuously arriving external input messages within three ranges of arrival rate. In the underuse and operational ranges, the latency fluctuates but the general tendency is stable. In contrast, the latency in the overload range tends to increase. This shows that the latency will then eventually reach infinity. And this confirms the theoretical analysis in Section 5.2.1.

5.3 Quantitative Analysis of Performance

In this section we present quantitative analysis of throughput and latency of RSPs. This will be used later as a guideline for the scheduler to optimise the performance. An RSP receives an infinite sequence of external input messages from the environment, processes them and sends out their outputs. When an external input message arrives and the RSP is too busy to take the message in, the message has to wait in the input queue. Based on this behaviour, an RSP can be considered as a queuing system. We consider here only stable systems where the arrival rate does not exceed the peak throughput and the number of external input messages inside the RSP $\overline{M_{cp}}$ is bounded. That is when the arrival rate is in the underuse and operational ranges.

5.3.1 Throughput Analysis

5.3.1.1 Uniformed Shared Memory Platforms

Consider an RSP deployed on a uniformed shared memory platform consisting of N homogeneous CPU cores for a time interval $P = [0, p]$.

After the time interval P, M external input messages have been completed and M_{cp} external input messages are partly processed. Let the average computational time required to complete one external input message be $\overline{C}$. The total computational time required to complete these M messages is $C_M = M \cdot \overline{C}$. The total computational time for partly processing these messages is C_{Mcp}. Since $\overline{M_{cp}}$ is bounded, M_{cp} is bounded and so is C_{Mcp}.

During the interval P, the total processing time of N cores is $\tau = N \cdot p$. The average idling time per core is W and the average overhead time per core is O. The relative idling time of the system is defined as $\widetilde{W} = \frac{W}{p}$. This is the average amount of CPUs which are idle. Similarly the relative overhead time is $\widetilde{O} = \frac{O}{p}$ and this is the average amount of CPUs on overhead work.

On shared memory platforms, stream transfers are simply memory access operations and therefore are negligible compared to node computations. It is plausible to consider that N cores spend the period P only for the computations of M completed messages; the computations of M_{cp} partly processed messages; and idling time. We have:

$$\begin{aligned}\tau &= N \cdot p \\ &= C_M + C_{Mcp} + W + O \\ &= M \cdot \overline{C} + C_{Mcp} + \widetilde{W} \cdot p + \widetilde{O} \cdot p\end{aligned}$$

Therefore,

$$M = \frac{N \cdot p - C_{Mcp} - (\widetilde{W} + \widetilde{O}) \cdot p}{\overline{C}}$$

The throughput over the period P is:

$$
\begin{aligned}
TP &= \frac{M}{p} \\
&= \frac{N \cdot p - C_{Mcp} - (\widetilde{W} + \widetilde{O}) \cdot p}{\overline{C} \cdot p} \\
&= \frac{1}{C} \cdot \left(N - \frac{C_{Mcp}}{p} - \left(\widetilde{W} + \widetilde{O} \right) \right)
\end{aligned}
$$

When the RSP processes infinite external input messages, the overall throughput is obtained when $p \to +\infty$. As C_{Mcp} is bounded, $\lim_{p \to +\infty} \frac{C_{Mcp}}{p} = 0$. Therefore, the overall throughput is:

$$
TP_{p \to \infty} = \frac{(N - \widetilde{W} - \widetilde{O})}{\overline{C}} \tag{5.1}
$$

5.3.1.2 Distributed Platforms

On shared memory platforms, the communication cost between nodes is negligible. This makes the throughput independent from where each task is executed. When deploying RSPs on distributed platforms, the communication cost becomes significant and the throughput formula in Equation 5.1 is no longer applicable.

Consider a distributed platform consisting of multiple PEs, each of which is a uniform shared memory platform. When deploying an RSP on such a platform, an intuitive approach is to divide the set of tasks of the RSP into multiple subsets, and assign them to separate PEs of the platform. Each PE has its own local scheduler for their assigned tasks. This approach only works for static RSPs with fixed structures during runtime. For dynamic RSPs, statistical observation of changing structure can be used to stabilise the RSP before applying this approach. In this section, we present a quantitative evaluation of throughput for distributed platforms.

A distributed platform is represented as an undirected graph $\mathcal{H} = (\mathcal{R}, \mathcal{L})$ where $\mathcal{R}$ is the set of vertices, also the set of PEs; and $\mathcal{L}$ is the set of edges, each of which is the communication link between PEs. This graph is called *target graph*. Each PE, R, has weight $w(R)$, which equates to its number of cores. A communication link between two processing elements R_i and R_j is denoted as $L_{R_iR_j}$ and has weight $w(L_{R_iR_j})$ which equates to its bandwidth in Byte/s.

As detailed in Section 2.4, each task is a representation for an instance of a node and they communicate with each other via uni-directional streams. The RSP therefore can be represented as a directed graph $\mathcal{G} = (\mathcal{T}, \mathcal{S})$ where $\mathcal{T}$ is the set of tasks, and $\mathcal{S}$ is the set of streams. This graph is called *task graph*. The computation weight of each task T

in $\mathcal{T}$ is defined as the average time that T requires to performs its contribution to an external input message. This amount of time depends on the PE onto which the task is mapped. The computation weight of task T on processing element R is notated as $w^R(R)$ and is measured in seconds.

A stream connecting two tasks T_i and T_j is denoted as $S_{T_iT_j}$. The stream also has a weight, called the communication weight $w(S_{T_iT_j})$, equal to the average amount of data to be transferred over $S_{T_iT_j}$ for the completion of an external input message. It is the average total size of messages that the stream $S_{T_iT_j}$ transfers during its contribution to each external input message. The unit of the stream weight is Byte/message. The values of $w^R(T)$ and $w(S_{T_iT_j})$ can be easily obtained using an appropriate monitoring framework, for example the one presented in Chapter 6.

A mapping configuration of an RSP $\mathcal{G} = (\mathcal{T}, \mathcal{S})$ over a distributed platform $\mathcal{H} = (\mathcal{R}, \mathcal{L})$ is defined as group of partitions MpC $= \{P_R \mid R \in \mathcal{R}\}$. The partition P_R is the set of tasks that are mapped to R, i.e. $(\forall R \in \mathcal{R} \quad P_R \subseteq \mathcal{T}) \wedge (\forall R_i, R_j \in \mathcal{R}, R_i \neq R_j \quad P_{R_i} \cap P_{R_j} = \emptyset) \wedge (\cup_{R \in \mathcal{R}} \quad P_R = \mathcal{T})$.

For shared memory platforms where we can ignore the message transfer; the completion of an external input message is formed by a set of node invocations spread over the contributions of all tasks. On distributed platforms, the cost to transfer messages on a stream within a partition can still be considered minor. In contrast, message transference on a stream across two partitions is costly. The completion of an external input message therefore spreads over both tasks on both partitions and the stream communications among them.

Partition P_R is considered to **complete** an external input message I_x when its tasks have completed their contributions to I_x. The communication weight between two partitions P_{R_i} and P_{R_j} is defined as the total weight of all streams across them. This represents the average amount of data to be transferred between the two partitions P_{R_i} and P_{R_j} within the completion of an external input message.

$$Comm(P_{R_i}, P_{R_j}) = \sum_{T_i \in P_{R_i}, T_j \in P_{R_j}} w(S_{T_iT_j}) \tag{5.2}$$

Since the completion of an external input message is stretched over all partitions and the communications among them, the throughput of the RSP is determined based on two kinds of throughput: the computation throughput and communication throughput.

The **computation throughput** of the partition P_R is the average number of external input messages that P_R completes within a time unit. Since each PE is a uniform shared

memory platform, Equation 5.1 can be used to derive the throughput of partition P_r as follows:

$$TP_{comp}(P_R) = \frac{w(R) - \widetilde{W_R} - \widetilde{O_R}}{\overline{C_R}} = \frac{w(R) - \widetilde{W_R} - \widetilde{O_R}}{\sum\limits_{T \in P_R} w^R(T)} \quad (5.3)$$

In this formula, $\overline{C_R}$ is the average time that all tasks in the partition contribute to completely process an external input message. This equals the total computation weight of all tasks in the partition. $\widetilde{W_R}$ and $\widetilde{O_R}$ are the relative idling time and the relative overhead time of the local scheduler of R, respectively.

The **communication throughput** between the two partitions P_{R_i} and P_{R_j} is amount of their communication weight that can be transferred via the physical link between the PEs R_i and R_j within a time unit. This is determined by dividing the bandwidth between PEs R_i and R_j by the communication weight between two partitions:

$$TP_{comm}(P_{R_i}, P_{R_j}) = \frac{w(L_{R_iR_j})}{Comm(P_{R_i}, P_{R_j})} \quad (5.4)$$

The communication weight $Comm(P_{R_i}, P_{R_j})$ is the amount of data that needs to be transferred between the two partitions P_{R_i} and P_{R_j} for the completion of an external input message. The communication throughput therefore can be considered as the number of external input messages that can be completed by the communication link.

Since the computation of partitions and communication among them can occur in parallel, the throughput of the RSP with a mapping configuration MpC is intuitively the minimum of all computation throughputs and communication throughputs.

$$TP(\text{MpC}) = \min_{R_i, R_j, R_k \in R} \left(TP_{comp}(P_{R_i}), TP_{comm}(P_{R_j}, P_{R_k}) \right) \quad (5.5)$$

5.3.2 Latency Analysis

According to Little's law [Lit61], the overall latency is equivalent to the average number of external input messages in the RSP, divided by the message consumption rate of the RSP:

$$L = \frac{\overline{M_{cp}}}{\lambda_{consumption}} \quad (5.6)$$

Where L is the latency and $\lambda_{consumption}$ is the rate at which the RSP consumes external input messages. $\lambda_{consumption}$ is also called the consumption rate.

Within a stable system, $\lambda_{consumption} = TP$. The latency in this case is:

$$L_{stable} = \frac{\overline{M_{cp}}}{TP} \tag{5.7}$$

5.4 Chapter Summary

This chapter has discussed the performance of RSPs in terms of throughput and latency. By analysing the effect of the message arrival rate on the performance, this chapter identified the borders of throughput and latency within different ranges of the arrival rate. Additionally, the chapter presented a quantitative analysis of throughput and latency, which is used in Chapter 7 and Chapter 8 for deriving efficient scheduling of RSPs on shared-memory multi-core platforms and distributed platforms.

Chapter 6

Monitoring of Reactive Stream Programs

In this chapter, we present a multi-level monitoring framework for RSPs on shared memory platforms. The monitoring framework provides information required to understand the RSP's runtime behaviour. We also show that the collected information is useful to calculate the performance metrics and give some potential guidelines for automatic load balancing, bottle detection and efficient scheduling strategies.

6.1 Conceptions of the Monitoring Framework

At the runtime system level, an RSP is represented by a set of runtime components connected by streams. At the execution layer, each runtime component later is wrapped into a task and sent to a scheduler. A typical scheduler includes two main components: a mapper and a time scheduler. The mapper decides where a task should be executed and the time scheduler decides when and how long.

In the following, we present the concept of our monitoring framework as shown in Figure 6.1. The framework collects the information from two levels: the runtime system and the execution layer.

6.1.1 Monitoring the Runtime System

At the runtime system level, the monitoring framework observes runtime components and messages to obtain the following information:

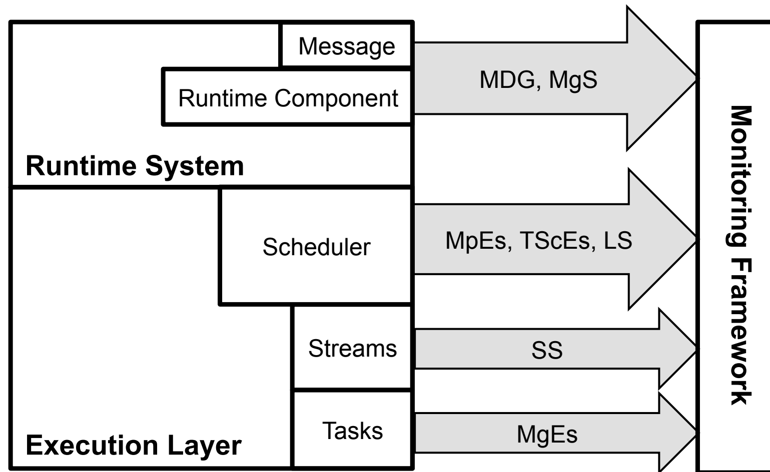


FIGURE 6.1: Monitoring framework

- **Message Derivation Graph (*MDG*).** To be distinguishable, each message on creation is assigned with a unique identifier (ID). While an RC consumes input messages and produces output messages, the monitoring framework observes the relationships of these messages. These relationships are used to create the MDG as described in Section 2.4.3.

- **Message Size (*MgS*).** When a message is sent to or created by the RSP, the monitoring framework obtains the size the message. The unit of message size is Byte.

6.1.2 Monitoring the Execution Layer

The monitoring framework observes three main objects of the execution layer including tasks, streams and the scheduler to obtain the following information:

- **Message Event (*MgE*).** When a task consumes or produces a message, the monitoring framework records this as a message event, i.e. message-consumed or message-produced, respectively. The event information comprises: the time it happens, the processing task, and the message involved.

- **Stream State (*SS*).** The monitoring framework observes every communication channel, i.e. stream, to memorise the task reading from the stream (called the *task reader*), the task writing to the stream (called the *task writer*), and the number of messages waiting on the stream (called the *fill level*). Together with the stream's maximum capacity, the fill level is used to determine if the stream is *full* or *empty*.

- **Mapping Event (*MpE*)**. A mapping event occurs when a task is assigned to a core. For these events, the monitoring framework records the event time, the core and the task.

- **Time Scheduling Event (*TScE*)**. A time scheduling event occurs when a task is created, destroyed, blocked, dispatched (i.e. sent to a core for execution), and yielded (i.e. halted while input messages are still available due to the scheduling policy). These events are denoted as *task-created*, *task-destroyed*, *task-dispatched*, *task-blocked*, and *task-yielded* respectively. The information of these events comprises the task identification, the event name and the time it has occurred.

- **Resource Load (*RL*)**. The monitoring framework keeps track of the workload on each core including execution time and idling time. Adding these two time values provides the response time.

6.2 Potential Benefits of the Monitoring Framework

As presented in Section 6.1, the monitoring framework provides seven kinds of information: Message Derivation Graph (*MDG*), Message Size (MgS), Message Event (*MgE*), Stream State (*SS*), Mapping Event (*MpE*), Time Scheduling Event (*TScE*) and Resource Load (*RL*).

These kinds of information can be used for different purposes including monitoring and optimisation. In this section, we present 4 potential usages from different combinations of the information (Table 6.1).

Information	*Performance Metric Calculation*	*Automatic Load Balancing*	*Bottleneck Detection*	*Extracting RSP Properties*
MDG	✓			✓
MgS				✓
MgE	✓			✓
SS			✓	✓
MpE		✓		
TScE	✓	✓	✓	✓
RL		✓		

Table 6.1: Monitoring information needed by different monitoring use cases

For convenience, we denote $reader(S)$ and $writer(S)$ as the task reader and the task writer of stream S. Also $size(M)$ is denoted as the size of message M. Since an RSP receives a virtually infinite sequence of external input messages from the environment, some variables are calculated based on a sufficiently large interval P. We define

$consumed_P(T)$ as the set of messages consumed by task T during the observed interval P; and $produced_P(T)$ as the set of messages produced by T during P. These two sets can be easily extracted from *MgEs*. The set of messages being processed by T during P is:

$$processed_P(T) = \{M|M \in consumed_P(T) \wedge d_successor(M) \subseteq produced_P(T)\}$$

$transferred_P(S)$ is defined as the set of messages which have passed over S during the observed interval P. These are messages produced by the writer of S and consumed by the reader of S during P.

$$transferred_P(S) = produced_P\,(writer(S)) \cap consumed_P\,(reader(S)) \tag{6.1}$$

During the interval P, a task T can be dispatched several times due to the scheduling policy and the availability of input. The execution of T during P is therefore spread over multiple intervals. Each interval is marked by 2 time scheduling events. The starting mark is an either *task-created* or *task-dispatched* event of T. The ending mark is the next either *task-destroyed* or *task-blocked* or *task-yielded* event of T. The sum of these intervals forms the execution time of T over interval P. This is denoted as $ET_P(T)$.

6.2.1 Performance Metric Measurement

The performance of RSPs is usually evaluated for individual nodes and for the overall RSP in terms of latency and throughput.

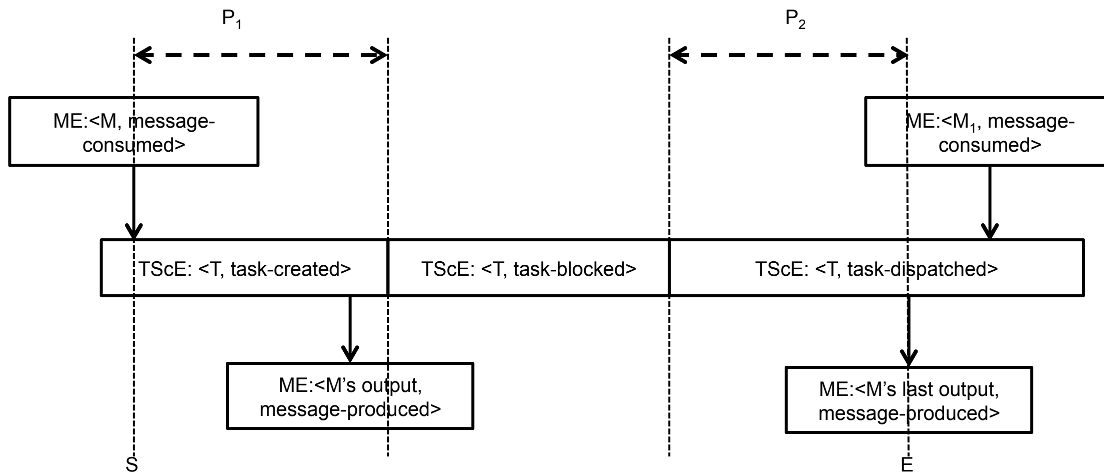


Figure 6.2: Latency of an individual task

Latency of an individual task T for processing a message M — $Latency(T, M)$. This is the execution time of T within the interval between two points: i) where T consumes M (S in Figure 6.2); and ii) where T produces M's last output message (E in Figure 6.2). The monitoring framework provides this information as follows. The *MDG* is used to find M's output messages, i.e. M's successors. Then appropriate *MgEs* are used to determine the two interval points: the event time S when M is consumed and the event time E when the last output message is produced. Finally, *TScEs* of T help to yield T's total execution time within the specified interval. In the example in Figure 6.2, the latency is the sum of P_1 and P_2.

Throughput of an individual task T — $TP(T)$. It is calculated by dividing the total execution time of T by the number of processed messages during the observed interval P.

$$TP(T) = \frac{|processed_T(P)|}{ET_P(T)}$$

Latency of an RSP for processing an external input message I — $L(I)$. This is the time interval from when I is consumed by the RSP until all its derived external output messages $derived_output(I)$ are produced. The *MDG* is used to calculate $derived_output(I)$. Then *MgEs* are used to determine when I is consumed and when the last external output message is produced.

Throughput of an RSP — TP. It is computed based on the number of external input messages the RSP has completely processed during the observed interval P. By using the *MDG* and *TScEs*, we can identify the set of external input messages consumed during P and we denote this set as EI_P. Similarly, the set of external output messages produced during P are denoted as EO_P. The number of completed external input messages is the cardinality of CEI_P which is the set of completed external input message within P. CEI_P is calculated as follows.

$$CEI_P = \{X | (X \in EI_P) \wedge (derived_output(X) \subseteq OI_P)\}$$

The throughput is calculated by dividing the cardinality of CEI_P by P:

$$TP = \frac{|CEI_P|}{P}$$

6.2.2 Extracting RSP Properties

In addition to performance calculations, the monitoring framework provides sufficient information to extract properties of an RSP including the computation weight of each

task and the communication weight of each stream. These two properties are used to analyse the throughput of RSPs on distributed systems as presented in Chapter 5.

The **computation weight** of a task T is the average computational time required for T to finish its contribution to an external input message. The computation weight of T is varied depending on which processing element it is executed on. It is common that the RSP is run on each processing element R of the distributed system to derive the computational weight for each task. The **communication weight** of a stream S is the average amount of data to be transferred over S to complete its contribution to an external input message. These two properties can be obtained by using the monitoring framework to observe the activity of the RSPs during an interval of P. The monitoring information is then used to derive these properties as follows.

Computation weight of a task T — $W^R(T)$. Given an external input message I, the computational time that T requires to complete its contribution to I is:

$$computation_time^R(T, I) = \sum_{M \in consumed_P(T) \cap successor(I)} Latency(T, M)$$

With EI_P is the set of external input messages consumed by the RSP during P, we compute the computation weight of T as follows:

$$w^R(T) = \frac{\sum_{I \in EI_P} computation_time^R(T, I)}{|EI_P|}$$

Communication weight of a stream S — $W(S)$. With an external input message I, the amount of data to be transferred over S during I's completion is:

$$data_size(S, I) = \sum_{M \in transferred_P(S) \cap successor(I)} size(M)$$

Where EI_P is the set of external input messages during P, the data weight of S is:

$$W(S) = \frac{\sum_{I \in EI_P} data_size(S, I)}{|EI_P|}$$

In addition to the above profiling statistics, the monitoring framework also allows the user to keep track of the graph of tasks, i.e. the structure of the RSP, by updating the relations between tasks and streams.

6.2.3 Automatic Load Balancing

Load balancing is a basic strategy to improve system performance by maximising resource utilisation. There are two types of algorithms for load balancing: static and dynamic. The static ones are applied before any input processing and require prior assumptions about runtime behaviour such as the response time of each task. The dynamic load balancing algorithms are different in that they use the system-state and are applied at runtime. For this reason, dynamic algorithms are a natural use case for our monitoring framework. In the following, we present two approaches of using the monitoring information to guide dynamic load balancing.

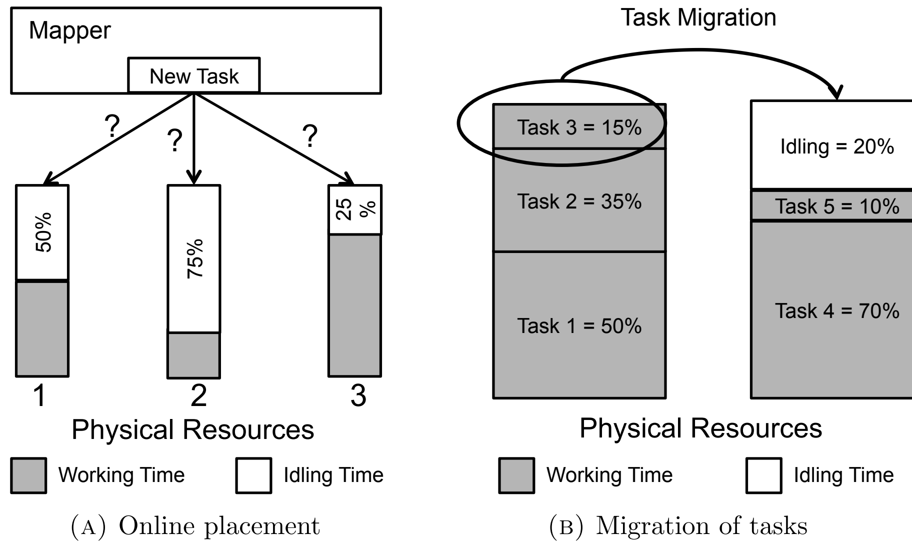
The first approach is the online placement balancing technique (also called centralised load balancing in [WA05]) in which new tasks are dynamically assigned to physical resources depending on the system state (Figure 6.3a). Using the monitoring framework, the system state can be expressed by the *RL* in terms of execution time and idling time. This information is used to implement the mapping policy: a task is assigned to the physical resource with the least execution time, i.e. most idling time. In the example in Figure 6.3a the second physical resource will be chosen for the new task. This simple dynamic mapping aims to balance the working load while minimising the idling time of physical resources.

The second approach is a task migration technique which controls the load balance by moving tasks among physical resources (Figure 6.3b). Many algorithms have been designed using this approach [ELZ86, SK90, SKS92, ZKS94, LSK97]. Algorithms of this approach usually have four components:

- *Information Policy*: specifies what information about the system-state is necessary and how to collect such information
- *Transfer Policy*: determines whether a physical resource should participate in a task migration
- *Location Policy*: identifies the suitable destination for the task migration
- *Selection Policy*: decides which tasks are eligible to migrate.

In the following, we define a simple instance of this approach using the monitoring information to define these policies:

- *Information Policy.* The collected information includes *RL* and *TScEs*, and the method of collecting is using our monitoring framework.



(A) Online placement (B) Migration of tasks

FIGURE 6.3: Deployment of automatic load balancing

- *Transfer Policy.* A physical resource PR_s should participate in a task migration if its current load is 100%, i.e. the idling time is zero.
- *Location Policy.* A physical resource PR_d should be a destination with the largest non-zero idling time.
- *Section Policy.* If there exists a PR_d and a PR_s, a task T is chosen from PR_s to migrate if T's current load is smaller than the idling time of PR_d. T's current load is calculated by using *TScEs* (discussed in Section 6.1.2).

In the example in Figure 6.3b, the first physical resource is busy all the time while the second one has a 30% idling time. Therefore, tasks from the first physical resource are migrated to the second one. Among the three tasks of the first physical resource, *Task 3* is the best candidate for the migration since only its workload is smaller than the idling time of the second physical resource.

6.2.4 Bottleneck Detection

Bottlenecks occur where the performance of a system is limited by a single task or a limited set of tasks (called bottleneck points). Knowledge of bottleneck points can help to improve performance by different mechanisms: for example, by assigning the higher scheduling priority to the bottleneck points so that they are scheduled more often.

In the following, we demonstrate a technique to detect bottleneck points by using *SSs* and *TScEs*. By intercepting the *task-blocked* state, *TScEs* of a task A can provide the blocking frequency of A. The reason for A being blocked is provided by the *SS* of the

communication streams between A and other tasks. In particular, the SS of a stream keeps track of alternations on the stream revealing the dynamic interrelation among the stream reader and stream writer. Consider a stream where A is the task reader, and B is the task writer. A is blocked by B if A tries to read from the stream while it is empty. In case of bounded streams, B is blocked by A when trying to write to the stream while it is full.

After obtaining the frequency of which a task blocks others, determining bottleneck points is straightforward. Tasks that cause high blocking frequencies to others tasks are considered bottleneck points.

6.3 Implementation of the Monitoring Framework in S-Net and LPEL

In this section, we show the implementation of the monitoring framework in the S-Net runtime system and LPEL execution layer. The implementation's overview is shown in Figure 6.4.

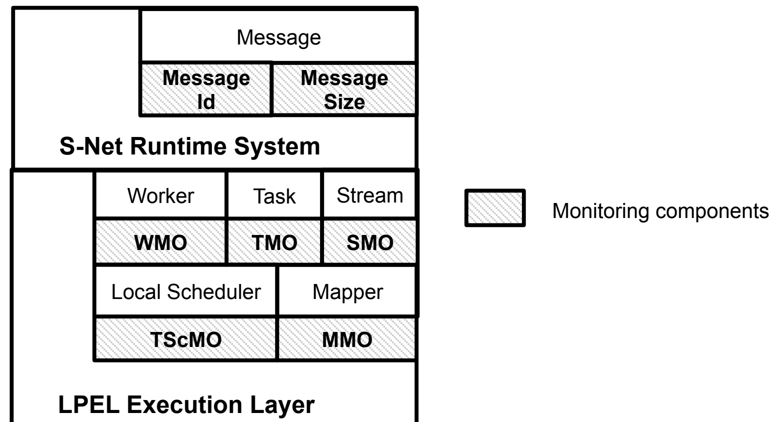


FIGURE 6.4: The monitoring framework implementation in S-Net and LPEL

6.3.1 Instrumenting the S-Net runtime system

Our goal is to obtain the monitoring information presented in Section 6.1 for the S-Net runtime system:

- **MDG**. Each node in the MDG is denoted as the message ID. When a message is created, the monitoring framework generates its ID. When the RSP is executed

on a distributed system, the message ID includes two parts. The first part is the unique index of the PE on which the message is created. The second part is the sequence number of the message within the PE. This sequence number is increased by one after generating an ID for each message. When the RSP is executed on a shared memory platform, the first part of the message ID is ignored. Edges of the MDG are constructed by the relationships between messages which can be obtained by observing the consumption and production activity of each RC. As each RC is wrapped into one task, these relationships can also observed in the execution layer. It is implemented this way for the simplification as it involves stream read and write activities which are under the control of the execution layer.

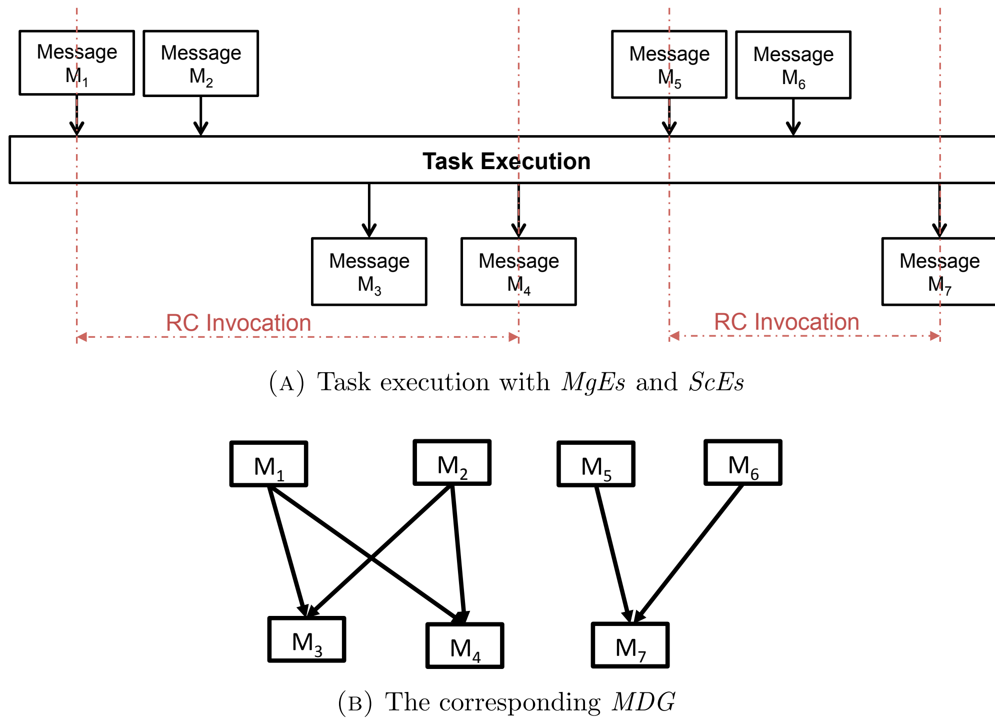
- **Message Size**. The message size is used to extract the communication weight of each stream. This weight represents the communication cost between RCs and is used to calculate the communication cost between PEs of the distributed system (Chapter 5). However, when deploying an S-Net program on a distributed system, the communication between PEs consists of not only the cost to exchange data but also the cost to send the references and to request for data in the case of flow inheritance (see Section 2.5.4). The monitoring framework therefore instrument the S-Net RTS before it invokes the MPI interface to send references, fetch requests and data. To obtain the accurate communication cost between each pair of RCs, the S-Net program needs to be run in a special mode where each RC of the S-Net program is mapped onto a separate logical PE. Each logical PE is a process on the same physical PE. Although sharing the same physical PE, the data communication pattern between RCs still hold and therefore their communication cost can be measured accurately.

6.3.2 Instrumenting the LPEL Execution Layer

As depicted in Figure 6.4, LPEL is instrumented with different monitor objects to provide *MgEs*, *SS MpEs*, *ScEs* and *RL* as follows:

- **Message Event**. Each task in S-Net is assigned with a unique identifier and is also equipped with a *Task Monitor Object (TMO)* as shown in Figure 6.4. The TMO monitors the task execution to catch two kinds of message events: *message-consumed* and *message-produced*. Whenever any of these events happens, the TMO records the time and the message information including the identifier and size.

  An S-Net task at runtime may process several messages within one execution (Figure 6.5a). However, information from message events can be used to construct the *MDG* without any extra information. This is because box executions for



(A) Task execution with *MgEs* and *ScEs*

(B) The corresponding *MDG*

FIGURE 6.5: Node executions within a task execution

different input messages are not interleaved, and with the *message-consumed* and *message-produced* events there is an unambiguous causality from input to output messages, expressed as directed edges in Figure 6.5b.

- **Stream State**. Each stream in S-Net is instrumented by a *Stream Monitor Object (SMO)* to memorise the identifiers of the task reader and the task writer. The SMO also keeps track of the number of messages inside the stream.

- **Mapping Event**. The LPEL mapper is instrumented by a *Mapping Monitor Object (MMO)* to capture mapping events. Each worker in LPEL is assigned a unique identifier. When a task is allocated to a worker, i.e. a mapping event occurs, the MMO records the identifiers of the task and the worker.

- **Time Scheduling Event**. Each LPEL worker has its own local time scheduler, which is instrumented by a *Time Scheduling Monitor Object (TScMO)*. When a task changes its state, i.e. a scheduling event occurs, the TScMO records the time, the task identifier and its new state. In LPEL, there are five task states: *task-created*, *task-blocked-by-input*, *task-blocked-by-output*, *task-resumed*, and *task-destroyed*.

- **Resource Load**. Each worker is exclusively mapped to a processor/core, which is considered to be an individual computational resource. Therefore, each worker is instrumented by a *Worker Monitor Object (WMO)*. The worker's WMO produces

the execution time by accumulating execution times of all its tasks. A worker becomes idle when it has no *ready* task. The WMO also observes these occurrences to form the worker's idling time.

6.3.3 Operation Modes

The implementation of the monitoring framework in LPEL and S-Net supports different monitoring flags to control the level of desired monitoring information.

- MAPPING_FLAG indicates mapping events are captured
- SCHEDULING_FLAG is set to catch time scheduling events
- STREAM_FLAG indicates SMOs are active to observe stream states
- MESSAGE_FLAG is set to record message events
- LOAD_FLAG is set to collect the resource load
- ALL_FLAG is an alias to set all other flags

If no flag is set, the application is executed as normal but without producing any monitoring information. Different flags can be combined for specific purposes.

6.4 Evaluation of the Monitoring Framework

The monitoring framework instruments the LPEL and the S-Net runtime system by placing control hooks to collect monitoring information. This causes some overhead compared to the original S-Net and LPEL implementation even if no information is collected. Currently, all monitoring information is sent to the file system and stored in log files. The overhead is evaluated experimentally in terms of response time and size of log files.

In the experiments we measure the overhead with different use cases by setting the following different flag combinations:

- **COM1**: no flag is set. This is used to measure the minimum overhead caused by monitoring controls without observing any events.
- **COM2**: the combination of MESSAGE_FLAG and SCHEDULING_FLAG is used for performance metric calculation (Section 6.2)

Application	*#MpE*	*#ScE*	*#Message*	*#Stream*
ANT	$278 \cdot 10^3$	$2.4 \cdot 10^6$	$1.15 \cdot 10^6$	$10 \cdot 10^3$
DES	$320 \cdot 10^3$	$4.7 \cdot 10^6$	$470 \cdot 10^3$	62
MC	$5 \cdot 10^6$	$33.8 \cdot 10^6$	$19.3 \cdot 10^6$	$5 \cdot 10^6$
RT	$1.8 \cdot 10^3$	$20 \cdot 10^3$	$23 \cdot 10^3$	100

Table 6.2: Application properties running on a 48-core machine

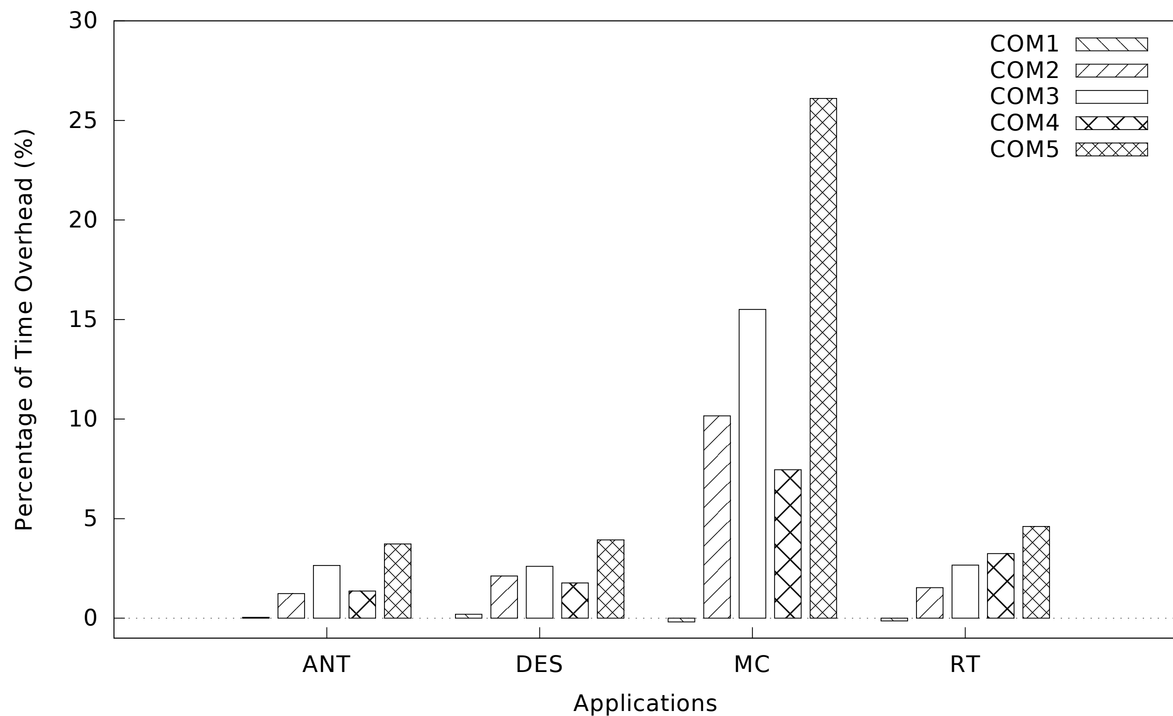
- **COM3**: MAPPING_FLAG, SCHEDULING_FLAG and LOAD_FLAG are set for automatic load balancing (Section 6.2)
- **COM4**: SCHEDULING_FLAG and STREAM_FLAG are combined to detect bottlenecks (Section 6.2)
- **COM5**: ALL_FLAGS is used to capture all events are captured, providing the maximum overhead

Note that the use case of extracting RSP properties is not included here as it is commonly used with the off-line mode, i.e. the RSP is pre-run with sample data to extract the properties which are analysed to generate a good mapping configuration for running with the real data.

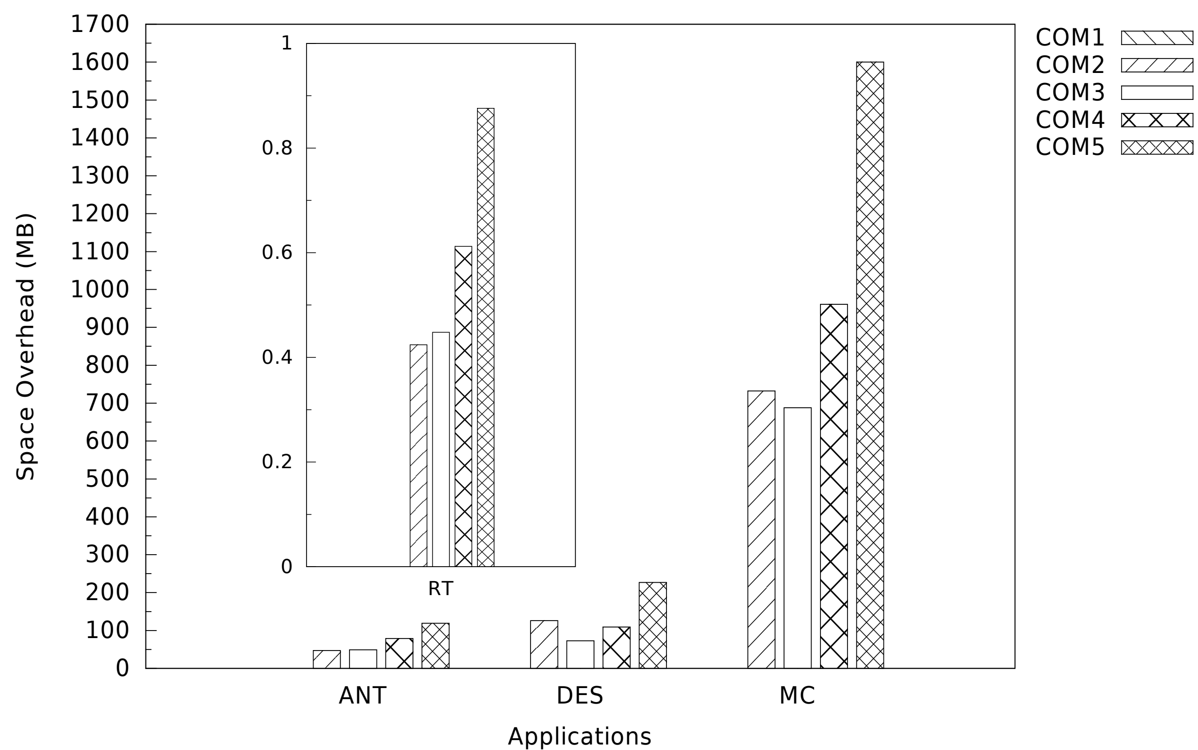
The monitoring overhead is caused by observing messages (TMO), streams (SMO), workers (WMO), the mapper (MMO), and the scheduler (TScMO). The overhead caused by observing messages and streams is proportional to the number of messages and streams, respectively. The overhead of MMO depends on the amount of mapping events while the overhead of TScMO and WMO depends on the number of scheduling events. These kinds of overhead are therefore affected by LPEL's scheduling policies. The experiment is performed on different applications ANT, DES, MC, and RT with various values of these variables as shown in Table 6.2. The details of these applications are presented in Chapter 4.

As shown in Table 6.2, MC has a very large number of tasks, each of which has very short life time. It also has an enormous number of scheduling events and messages. This is because the implementation of MC is too fine grained. Usually this kind of implementation is not sufficient in stream programming. MC is chosen for this experiment to show the overhead in an extreme case.

All applications are run on a 48-core machine comprising of 4 sockets with a 12-core AMD Opteron 6174 and a total of 256 GB main memory. The time and space overhead is shown in Figure 6.6. Generally, the time overhead depends on the number of monitored events. The minimum overhead (COM1) is negligible for most of the cases. There is even a negative overhead for MC and RT. We attribute this negative overhead to scheduling anomalies similar to timing anomalies in processors [WKPR05].



(A) The time overhead



(B) The space overhead (in MB)

Figure 6.6: The overhead of the monitoring framework in time and space

The time overhead of different combinations of flags varies from application to application depending on the number of monitored objects (mapping events, scheduling events, messages and streams). In the current implementation, the monitoring information is sent to the file system and therefore the time overhead is also affected by the response time of the application. As the file I/O is performed asynchronously by the operating system, the time overhead is quite small for a relatively long running application. Consequently, for most of the applications the time overhead is relatively small for all flag combinations. For the MC application the overhead is quite large because it has a very large number of monitored objects while the response time is small compared to the amount of data which outweighs any benefits of asynchronous I/O operations.

The space overhead is proportional to the amount of collected data, or the number of monitored objects. As shown in Figure 6.6b, MC with the highest number of monitored objects has the highest space overhead. Similarly, RT has the least space overhead as it has the smallest number of monitored objects.

6.5 Chapter Summary

The support of monitoring is essential for achieving high system utilisation of parallel execution platforms. In this chapter we presented a monitoring framework that is geared towards RSPs to monitor data for use cases including the calculation of performance metrics, extraction of RSP properties, automatic load balancing, and bottleneck detection. This monitoring framework extracts information from both, the runtime system as well as from the underlying execution layer.

The extracted information provides the trace of non-deterministic behaviours of the application at both levels. The monitoring approach is fully transparent to the user and is purely software based. The overhead of different monitoring scenarios is given in Figure 6.6, which shows for most benchmarks a negligible overhead of less than 5%. Only in the MC benchmark the overhead reaches up to about 26%, which is explained by the fact that in this benchmark the concurrency is too fine grained to be efficiently exploited using stream programming.

Chapter 7

Exploiting the Properties of RSPs for Efficiently Scheduling on Uniform Shared Memory Platforms

Based on the performance analysis in Chapter 5, this chapter presents a guideline for optimising both throughput and latency of RSPs. We use this guideline to design two novel heuristic schedulers. As centralised approaches, these schedulers exploit the properties of RSPs to define the task priority. The first one uses the notion of data demands on stream communications while the second one takes advantage of the structural position of tasks in the RSP. Since these features can be observed during runtime, these scheduling approaches require no assumptions about the RSP properties. They therefore are applicable to general RSPs with properties shown in Table 2.2. Particularly, they support general RSPs where the node computation behaviour can be variable and the program structure can be dynamic while this is a particular challenge for static scheduling based on formal constraints or probabilities.

7.1 Guidelines For Scheduler Design

This section presents guidelines to design a scheduler aiming for performance optimisation. The guidelines are based on the quantitative analysis of the performance discussed in Chapter 5.

7.1.1 Throughput Optimisation

According to Chapter 5, the throughput of an RSP is equivalent to the arrival rate λ in the underuse and operational ranges when $\overline{M_{cp}}$ is bounded. In the overload range when $\overline{M_{cp}}$ becomes infinity, the system gets overloaded while the throughput does not exceeds its peak value. We therefore focus only on the underuse operation ranges.

According to Equation 5.1, we have the following formula or the throughput:

$$TP = \frac{(N - \widetilde{W} - \widetilde{O})}{\overline{C}}$$

Since $\overline{C}$ is the mean required computation time to completely process one external input messages, $\overline{C}$ depends on the implementation and the underlying hardware. These factors are not under the sphere of control of the scheduler. To optimise the throughput the scheduler therefore should: i) keep $\overline{M_{cp}}$ bounded and ii) reduce the relative idling time $\widetilde{W}$ and the relative overhead time $\widetilde{O}$.

7.1.2 Latency Optimisation

Since the latency becomes infinity in the overload range, we do not consider this range here. The latency optimisation is focused on two other ranges where $\overline{M_{cp}}$ is bounded. According to Equation 5.6, we have:

$$L = \frac{\overline{M_{cp}}}{\lambda_{consumption}}$$

To reduce the overall latency, the scheduler needs to increase the consumption rate and at the same time keep $\overline{M_{cp}}$ low. Within stable systems, the consumption rate is equivalent to the throughput, therefore to maximise throughput is also to contribute towards minimising the overall latency.

7.2 Heuristic Scheduling Strategies for Performance Optimisation on Symmetric Processors

In this section we propose a scheduler aiming to optimise the performance of RSPs on a uniform shared memory platform. The proposed scheduler employs heuristic strategies based on the above guidelines. Maximising throughput means maximising $\lambda_{consumption}$ and therefore contributing towards minimising the overall latency. For this reason, the

propose scheduler first tries to reach the optimal throughput by using a centralised approach. With the maximised throughput, the scheduler then attempts to minimise the latency.

Consider a shared memory platform of homogeneous physical resources, each of which is a CPU core. To deploy an RSP on such platforms, a scheduler consists of two sub-schedulers: a space scheduler which decides on which CPU core a task should be executed; and a time scheduler which decides when a task is executed and for how long. This section presents a scheduler based on the above guildlines to optimise both throughput and latency by minimising $\widetilde{W}$, $\widetilde{O}$ and $\overline{M_{cp}}$.

7.2.1 Space scheduler

In the proposed scheduler, we can consider one CPU core as a worker. The terms core and worker are used interchangeably in the rest of the chapter.

The space scheduler does not permanently map tasks to any worker. Instead ready tasks are stored in a central queue (CTQ). A task is assigned to a worker whenever it is free. Dynamic program structures are well supported by using the CTQ with its dynamic scheduling of tasks to available resources. That helps to reduce the relative idling time $\widetilde{W}$ but does not guarantee to minimise it. This depends on the time scheduler which controls the availability of ready tasks. This design of the space scheduler allows flexibility for the time scheduler to control the availability of ready tasks as well as adjust the value of $\overline{M_{cp}}$.

7.2.2 Time Scheduler

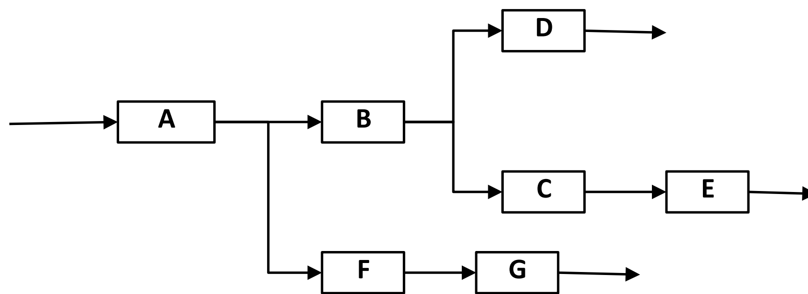
One responsibility of the time scheduler is to take a relevant ready task from the CTQ to be executed by a free worker, i.e. to define the task priority. Another responsibility is to decide for how long a worker should execute the assigned task, i.e. to define the scheduling cycle. For RSPs, it is hard to derive an exact scheduling policy providing the best performance because of their dynamic properties such as dynamic program structures and variable node behaviour.

The time scheduler on one hand has to activate enough ready tasks to keep the relative idling time $\widetilde{W}$ low; and on the other hand it has to control $\overline{M_{cp}}$. Note that the availability of ready tasks is also implied by the availability of messages inside the RSP. In the following, we propose two heuristic strategies for the task priority function which decides when a task should get executed. Tasks with higher priorities will be executed first.

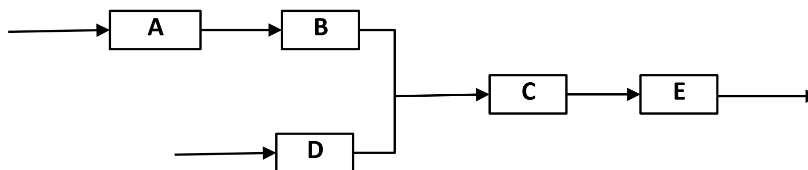
7.2.2.1 Position-based Task Priority

This heuristic decides the task priority by its structural position in the RSP. The general idea is a task that is closest to the exit tasks should be executed first. As all streams are uni-directional, this heuristic intuitively moves messages forward to the exit tasks as soon as possible, and therefore keeps $\overline{M_{cp}}$ low.

There are two ways to design of this heuristic. In the first design, the priority of a task is defined as the inverse of its distance to the exit tasks. The distance is defined as the number of tasks a message and its successors need to pass over until being sent out to the environment. This equals the number of vertices on route from the current task to an exit task on the task graph.



(A) An RSP with multiple exit tasks



(B) An RSP with multiple entry tasks

FIGURE 7.1: Examples of RSP with multiple entry tasks and multiple exit tasks

This design requires the RSP to have only one exit task. Otherwise, it is unable to determine the priority for tasks from which there are multiple routes to different exit tasks. Consider the example shown in Figure 7.1a, an output message from B can be sent either to the exit task D or to C and then to the exit E. As the behaviour of B can be variable, it is not feasible to decide to use the distance from B to D or the distance from B to E as its priority. This design is also restricted to RSPs with a single entry task. In the case where there are more than one entry tasks with different distances to the exit task, this heuristic design can cause starvation and live-lock. Consider the example shown in Figure 7.1b, there is one exit task E and there are two entry tasks A and D continuously receiving messages from the environment. With the heuristic priority biased toward the exit task, we have task priority as: $Priority(E) > Priority(C) > Priority(B) = Priority(D) > Priority(A)$. Consider an application deployed in a platform with three

workers. When D constantly receives external input messages from the environment, three tasks D, C, E are constantly activated and occupy all the workers. That means inputs entering to A will not be processed. In the case where there is only one worker and C requires messages from both B and D before proceeding, live-lock occurs. This is because D has a higher priority than A and therefore it is constantly executed and sending messages to C. Meanwhile C can not be executed because it needs messages from B which depends on the execution of A.

The second design defines a task's priority as its distance from the entry tasks. As with the first design, this is also restricted to RSPs with a single entry task. Unlike the first one, this design support RSPs with multiple exit tasks as there is no task depending on their outputs. Therefore starvation and live-lock do not occur in this case.

Both of the above designs are restricted to RSPs with single entry tasks and also disfavour tasks close to the entry task. The entry task has the lowest priority and will not be scheduled until no other task is ready. This can lead to the scenario that increases the value of $\widetilde{W}$. That is when two or more workers finish their tasks at the same time and the entry task is the only ready task. Only one worker will get the entry task and others have to wait.

7.2.2.2 Demand-based Task Priority

This heuristic is based on the positive demand S_I and the negative demand S_O, where S_I is the total number of messages in the input streams and and S_O is the total number of messages in the output streams. The heuristic is proposed as follows.

- **The priority of an entry task should have a negative correlation with its S_O.** Entry tasks are ready as soon as there are external messages. Their executions create input for following tasks and make these tasks ready. This heuristic helps entry tasks to be executed when the potential of ready tasks is low. Once executed, their priority is reduced and after a certain time they have to release resources for other tasks. This keeps $\overline{M_{cp}}$ bounded.
- **The priority of exit tasks should be higher than other types of tasks.** This is because exit tasks send messages to the external environment, they should be executed as soon as possible to keep $\overline{M_{cp}}$ as low as possible.
- **The priority of a middle task should have a positive correlation with S_I and negative correlation with S_O.** Exit tasks should be executed as soon as possible, however they become ready only when messages are transferred over

the RSP passing other middle tasks. A middle task T_0 while performing an RC invocation consumes n messages from its input streams and produces m messages to its output streams which are read by other tasks $T_i|1{\le}i{\le}n$. The S_I value of T_0 is reduced by n and the S_O value of T_0 is increased by m. With this heuristic, the priority of T_0 is reduced and its chance to hold physical resources is reduced. Meanwhile the S_I values of tasks $T_i|1{\le}i{\le}n$ are increased. That means tasks $T_i|1{\le}i{\le}n$ will have a higher chance to be scheduled and the newly created messages are likely to move forward to the output.

7.2.2.3 Scheduling Cycle

Ideally each task after executing for a period should be returned so that other higher priority tasks can proceed. Task switching can cause overhead and locality loss. Therefore the worker should run a task long enough so that the task switching overhead becomes negligible. We propose a heuristic strategy to define the scheduling cycle based on a timeout value E_{sc}. Once assigned to a worker, a task is executed until it is blocked or the timeout value has been reached. The timeout value E_{sc} can be defined based on the number of RC invocations, the number of produced output messages or a time period. It is hard to analytically derive the value of E_{sc}. We therefore propose to derive this value through practical experiments.

7.2.3 Scheduling Design Comparison

We have presented above a centralised scheduling approach with two different heuristics for defining the task priority. These two heuristics aim to optimise both throughput and latency by minimising $\widetilde{W}$, $\widetilde{O}$ and $\overline{M_{cp}}$. Here we compare our scheduling design with other alternatives.

As discussed in Section 3.2, there are two general classes of schedulers: offline and online. Although offline schedulers often provide nearly optimal results, they require complete knowledge of the system at compile time. Our work here targets general RSPs with variable node behaviours and dynamic program structures. Therefore, offline schedulers are not suitable to solve our problem. We then focus on different scheduling approaches of online schedulers which make decisions at runtime based on the current state of the system.

As presented in Section 3.2, there are three main techniques to design an online scheduler: centralised, distributed and centralised mediation. Our scheduling approach is designed as an online scheduler where tasks are not statically assigned to any PE but are stored in

a central task queue. Two different heuristics are proposed to define task priority which decides the order of task execution. Since all tasks are stored together in the central task queue, the task distribution is based purely on the task priority.

In contrast, the distributed scheduling and centralised mediation techniques do not store all the tasks centrally. The decisions regarding task distribution are therefore affected not only by the task priority but also by the cost of relocating the task from the sender PE to the receiver PE. Consequently, the distributed scheduling technique is not appropriate to highlight the benefits of using the task priority to optimise the performance which is the focus of our work. However, the heuristics for defining the task priority can be still useful in these scheduling techniques when combined with other relevant policies.

7.3 Implementation of the heuristic priority functions

The proposed stream schedulers are implemented as new schedulers for the execution layer LPEL to support S-Net programs. LPEL was chosen for supporting task reallocation among CPU cores on a shared memory platform without extra cost; and for providing a sufficient mailbox implementation for core-to-core communications.

As presented above, the space scheduler does not permanently map tasks to any worker. Instead ready tasks are stored in a central queue (CTQ) and the order of tasks depends on their priority. When a worker is free, the task with highest priority will be sent to the worker. There are two alternatives of how to implement it. The first one is to dedicate a worker that acts as the **conductor** to manage the CTQ. Free workers will need to communicate with the conductor to request for a new task. The second one is to allow all workers to access the CTQ and to provide a locking mechanism to enforce mutual exclusion access.

Compared to the first option, the second one has some disadvantages. First, it requires workers to be interrupted for updating the task priority as the task priority can be varied during runtime. Second, any change of task status or task priority will require to update the CTQ. Since only one worker can access the CTQ at a time, updating the CTQ causes to prolong this exclusive period. This therefore increases the chance of worker contention and also extends the contention period.

In contrast, with the second option of implementation, the conductor always keeps the CTQ up-to-date and therefore the response to a task request from workers takes constant time. In fact, when receiving a task request, the conductor retrieves the task with highest priority in a constant time, and sends to the requesting worker. While

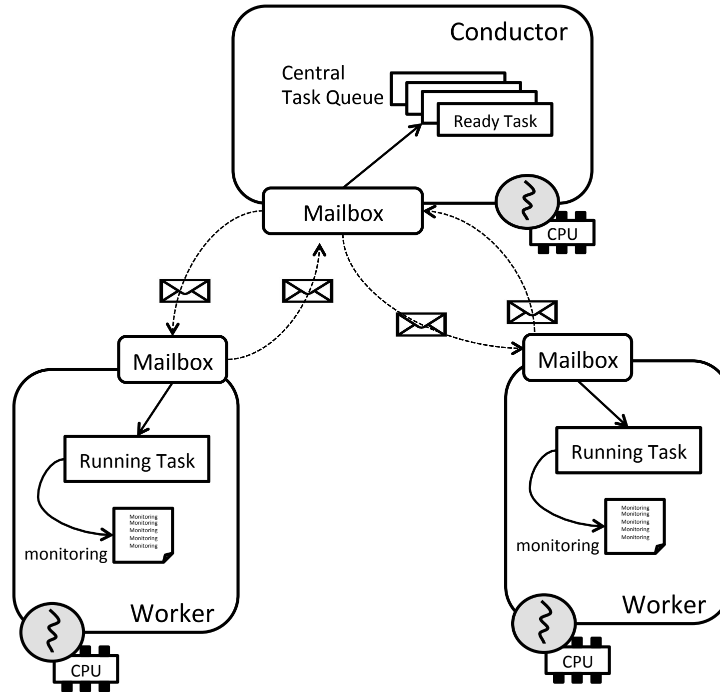


FIGURE 7.2: Design of heuristic stream schedulers for performance optimisation

the worker is working on the new task, the conductor is free to update the CTQ. Utilising the conductor also provides the flexibility for updating the task priority without interrupting workers. In addition, this way of implementation brings open directions for efficient communication protocols between the conductor and workers to avoid the bottleneck. These directions will be presented with more detail in Chapter 10. Although the dedicated conductor can be considered as scheduling overhead, this overhead can be paid off with a large number of cores.

For these above reasons, we use the first option to implement our proposed schedulers. The design of our schedulers is demonstrated in Figure 7.2. As the task priority is dynamically changing over time, one worker is dedicated as the **conductor** to keep track of the task priority. The conductor also arranges ready tasks according to their priorities by using a heap structure. The conductor is responsible to update the task priority when necessary. Once a worker is free, it requests a new task from the conductor. The conductor then chooses the ready task with the highest priority from the CTQ and sends to the requesting worker. All the communication between the conductor and workers is exercised via mailboxes.

Since S-Net constructs the RSP by a hierarchical combination of SISO operators, S-Net programs have single entry task and single exit task. Both the task priority functions are therefore feasible for S-Net. The rest of this section presents the implementation of two task priority functions. These functions are integrated into the centralised scheduling

approach to form new schedulers. We denote the scheduler with position-based task priority as **CS-pbp**, and the scheduler with demand-based task priority as **CS-dbp**.

7.3.1 Position-based Task Priority Function

To implement the position-based priority, one needs to know the structural position of a task within the RSP. This information could be obtained from the generic stream program information at the LPEL layer by keeping track of the task-stream relation. However, doing so is not feasible in case of potentially dynamic program structures, as this would result in a dynamic change during runtime of the task-stream relation as well. Keeping track of these relations can be very costly and would outweigh the benefits.

In this section, we present an alternative implementation that retrieves information about the RSP structures from the runtime system of the coordination language S-Net. This information is used to calculate the distance-based priority. The principle behind this technique is the fact that S-Net programs are always structured such that sub-networks with single input and output (SISO) are composed by operators which preserve this SISO property. Based on these structured network compositions the S-Net runtime system can maintain explicit information at each RC about its relative path through the network. We use this information on each S-Net RC as its distance from the entry. As introduced in Chapter 2, each S-Net RC is wrapped in a LPEL task. The terms RC and task therefore can be used alternatively

To provide the structural position of every task, S-Net associates each task with a data structure, called a location vector, describing the path from the entry task to the task itself. A location vector is a list of elements, each of which is a tuple of a combinator type and an index number. The values and meaning of these elements are shown in Table 7.1. For example, $\langle P, 2\rangle$ indicates the second operand of a parallel composition; $\langle R, 3\rangle$ specifies the third instance of a serial replication. Note that serial and parallel compositions can have more than two operands, for example `A | B | C` is a parallel composition of three operands A, B, and C. In the case where the S-Net program contains only one box, it can be considered as one serial composition with one operand.

Combinator Type		Index Number
S	Serial Composition	Operand Index
P	Parallel Composition	
R	Serial Replication	Instance Index
I	Parallel Replication	

Table 7.1: Values of a location vector element

As presented in Chapter 2, during runtime each parallel composition is represented by a pair of operators: a parallel compositor and a collector. There are indexed as 0 and ∞ respectively to indicate the beginning and ending operator of the composition. It is implemented similarly for parallel replication. The implementation of serial replication is more complicated as it has multiple serial replicators generated dynamically. They are indexed together with the operands as demonstrated in Figure 7.3. The first serial replicator and the collector are indexed with 0 and ∞.

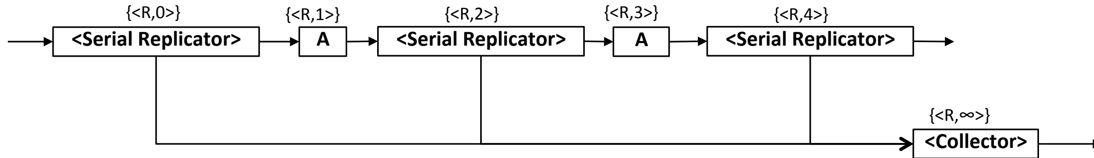


FIGURE 7.3: Runtime components with location vector of $A \star \{\langle stop \rangle\}$

A location vector $\{\langle T_1, N_1 \rangle, \langle T_2, N_2 \rangle \ldots \langle T_k, N_k \rangle\}$, where T_i is a combinator type and N_i is an index number, specifies an RC which is the operand/instance indexed N_k of the combinator T_k; and T_k is the operand/instance indexed N_{k-1} of the combinator T_{k-1}; and so on. Figure 7.4 shows location vectors of RCs in the Image Filter Application. At the top level, the application is formed by a serial composition with 3 operanDS. The first operand is box *Splitter* with location vector $\{\langle S, 1 \rangle\}$, and the third operand is box *Merger* with location vector $\{\langle S, 3 \rangle\}$. The second operand is a parallel replication represented by a parallel replicator $\{\langle S, 2 \rangle, \langle I, 0 \rangle\}$ and a collector $\{\langle S, 2 \rangle, \langle I, \infty \rangle\}$. Three operands of the parallel replication are instances of box *Filter* with location vector $\{\langle S, 2 \rangle, \langle I, 1 \rangle\}$, $\{\langle S, 2 \rangle, \langle I, 2 \rangle\}$, $\{\langle S, 2 \rangle, \langle I, 3 \rangle\}$ respectively.

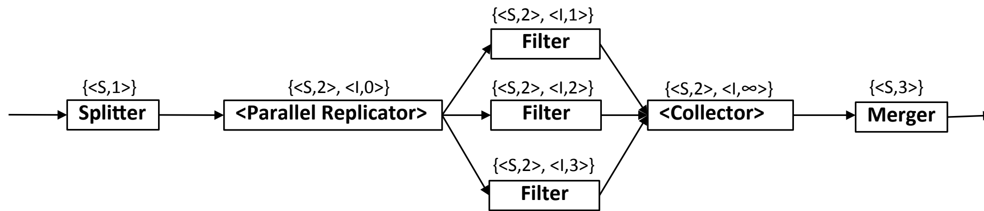


FIGURE 7.4: Runtime components with location vectors of the image filter application

A location vector of a task T shows the tracking from the entry task to T. Thus the distance d_{entry} from the entry task can be deduced from its location vector. Although it is not easy to derive a concrete value for the distance, it is trivial to compare the distance between two tasks. Before explaining how to compare d_{entry} of two tasks, we present 4 rules to compare d_{entry} of two location vector elements.

- When comparing operands/instances of a serial composition or serial replication, the one with larger index has a higher d_{entry};

- When comparing operands/instances of a parallel composition or parallel replication, they have the same d_{entry};
- Parallel compositor $\langle P, 0\rangle$ and parallel replicator $\langle I, 0\rangle$ always have a lower d_{entry} than their operands;
- The collectors of parallel composition $\langle P, \infty\rangle$ and parallel replicator $\langle I, \infty\rangle$ always have a higher d_{entry} than their operands.

To compare the d_{entry} between two tasks, one can traverse along their location vectors and compare each pair of elements. If these elements have the same d_{entry}, the process continues to the next pair. Otherwise, comparison of the last pair returns the final result. In the case where the process reaches the end of one location vector before the other, the one with larger size has the higher d_{entry}.

7.3.2 Demand-based Priority Function

According to Section 7.2, the priority of a middle task should have a positive correlation to its S_I and a negative correlation to its S_O. Table 7.2 lists some priority functions for middle tasks according to the proposed heuristic in Section 7.2. Functions PF_1 and PF_4 are simple and typical for functions with the same significance of S_I and S_O. Function PF_2 is an example for which S_I has higher significance and function PF_3 is an example for which S_O has higher significance. The priority function for entry tasks is the same for middle tasks but with S_I being zero. As an exit task (with $S_O = 0$) should have higher priority than other tasks, there are two choices. The first is to use the priority function of middle tasks but with S_O is zero; this makes an exit task a higher priority compared to a middle task with the same S_I value. The second is to set the priority of exit tasks to infinity ($+\infty$).

We carried out experiments with all the combinations of these priority functions for entry, exit and middle tasks. None of them has shown superior performance compared to the others. In fact, the variation coefficient is relatively small, about 2∼3%. For its simplicity, we choose to use the priority function PF_4 for middle tasks, for entry tasks with $S_I = 0$, and for exit tasks with $S_O = 0$.

To obtain instant values of S_I and S_O during runtime, the scheduler is supported by the stream monitoring framework presented in Chapter 6. This monitoring framework allows us to observe the fill level of streams (i.e. the number of messages currently in the stream). As the program structure is dynamic, a task's input and output streams are dynamic. This monitoring framework also allows us to keep track of this information.

Priority Function	T_{Middle}
PF_1	$\frac{S_I+1}{S_O+1}$
PF_2	$\frac{(S_I+1)^2}{S_O+1}$
PF_3	$\frac{S_I+1}{(S_O+1)^2}$
PF_4	$S_I - S_O$

For the first three priority functions, '1' is added to S_O to avoid division-by-zero. '1' is also added to S_I to have a fair proportion against S_O.

TABLE 7.2: Priority functions of the middle tasks

From the stream state of the monitoring framework, we can derive the task-stream relations and therefore calculation of the S_I and S_O values of each task is trivial. Note that, only local information about task-stream relations is required, the whole structure of the RSP is not necessary. In addition, the monitoring framework provides the other required information to analyse the throughput and latency of RSPs.

Although a task's priority changes whenever its S_I or S_O values changes, it is inefficient to constantly re-evaluate the priority and update its order in the CTQ. In this implementation, the task priority is updated in the two following situations.

- A task's priority is re-evaluated when it becomes ready and is added to the CTQ
- When a task T is halted (because either it is blocked, or it terminates, or it has finished its scheduling cycle), the priority of ready tasks that have stream connections with T is updated.

7.4 Evaluation: CS-dbp vs CS-pbp

In this section we compare the two priority functions based on the heuristic strategies CS-dbp and CS-pdp. The comparison is carried out in conceptional, implementational, and experimental matters.

7.4.1 Conceptional Comparison

Both of the schedulers are similar in that they use a quantitative analysis to optimise throughput and latency. To maximise the throughput, they both attempt to bound $\overline{M_{cp}}$

and cut down $\widetilde{W}$ by using the central approach. To minimise the latency, they both try to minimise the $\overline{M_{cp}}$.

To obtain these goals, while the CS-dbp uses the task's computational demand, the CS-pbp employs the structural information. Although both the computational demand and the structural information can be obtained by monitoring, it is easier and cheaper to get the former one. To acquire the computational demand of each task, only local information is needed. That includes the set of input streams, the set of output streams and the fill levels of these streams. In contrast, to obtain the program structure, the monitoring framework needs a global picture of task-stream relations. In addition, it needs to keep track of these relations while they are dynamically changed when the structure of the program is altered.

A cheaper way to obtain the structural information is to initialise the program structure from the source code; and then to rely on the RTS to maintain changes in the program structure during runtime. However, this technique spoils the separation between the execution layer and the RTS. That means the execution layer is no longer portable for other stream languages.

Furthermore, the position-based priority is only applicable to a smaller class of RSPs. It is restricted to RSPs with a single entry task if using the distance to the entry task metric. To use the distance to the exit task, the heuristic strategy requires RSPs to have both a single entry task and a single exit task.

7.4.2 Implementation Comparison

By using different priority functions, the conductor's response time varies and therefore can cause different worker waiting times. For CS-dbp, the task priority is dynamic, i.e. it varies during runtime depending on the task's positive demand S_I and negative demand S_O. When a task is executed, it changes the S_I and S_O not only of the task itself but also of its neighbours (tasks which it has a stream connection with). The conductor then needs to update the priority for all these involved tasks.

For CS-pbp, the priority of a task is unchanged during its lifetime and the conductor does not need to update it. However, unlike in CS-dbp, the task priority in CS-pbp is not a single numeric value but a vector. Comparing location vectors is more expensive.

7.4.3 Experimental Comparison

7.4.3.1 Experiment Set Up

The experimental comparison between two schedulers is performed on different S-Net applications including:

- DES: performs DES encryption on 32KB-size messages
- FFT: computes the FFT algorithm on messages of 2^{20} discrete complex values
- HIST: calculates a histogram of images with an average size of 5342 × 3371
- IMF: applies a series of filters on images with average size of 4658 × 3083
- OBD: detects 4 different types of objects from 1920 × 1080 images

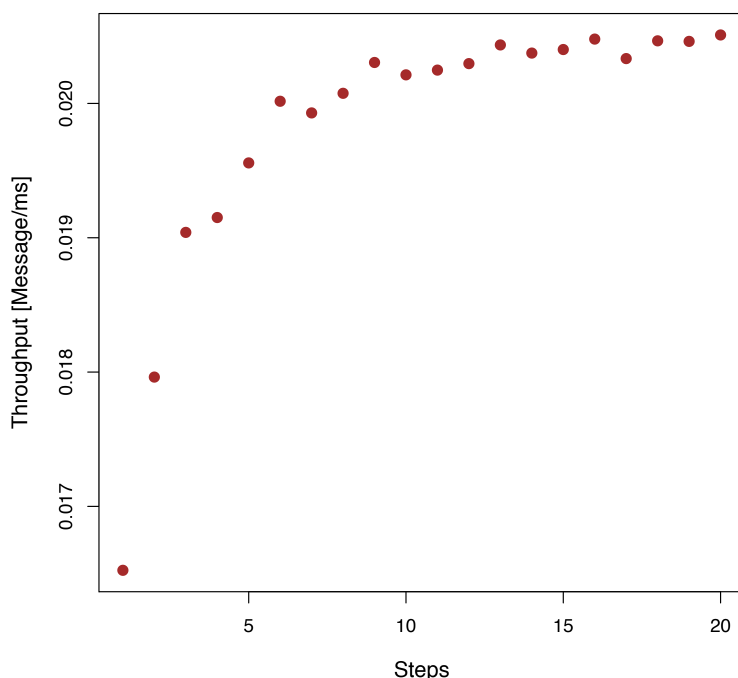


FIGURE 7.5: Throughput convergence of IMF on CS-dpb when increasing the number of external input messages

Each application is composed of a sub-network that performs the application's main function, and a parallel replication to create multiple instances of this sub-network. The number of instances depends on the number of cores. Therefore, an application will have more tasks when being executed with more cores. The details of these applications are shown in Chapter 4.

Benchmark	RSD_{step}	NoM	$PD_{Throughput}$	$PD_{Latency}$
DES_2	0.02	2000	0.02	0.09
DES_4	0.12	4000	0.15	0.32
DES_8	0.14	8000	0.02	0.08
DES_{16}	0.05	16000	0.01	1.37
DES_{32}	0.00	32000	0.04	0.28
DES_{46}	0.01	46000	0.04	0.72
FFT_2	0.07	3600	0.29	0.23
FFT_4	0.08	7200	0.23	0.09
FFT_8	0.05	14400	1.97	2.64
FFT_{16}	0.05	28800	1.15	1.04
FFT_{32}	0.02	57600	2.0	2.69
FFT_{46}	0.05	82800	0.66	3.85
$HIST_2$	0.05	2400	0.24	0.21
$HIST_4$	0.06	4800	0.95	0.99
$HIST_8$	0.05	9600	0.27	0.43
$HIST_{16}$	0.04	19200	0.14	0.6
$HIST_{32}$	0.05	38400	0.58	0.33
$HIST_{46}$	0.08	55200	1.02	0.88
IMF_2	0.03	2000	0.29	0.26
IMF_4	0.10	4000	0.73	0.7
IMF_8	0.08	8000	0.08	0.24
IMF_{16}	0.10	16000	0.58	0.7
IMF_{32}	0.11	32000	0.37	1.32
IMF_{46}	0.08	46000	0.45	1.82
OBD_2	0.02	2000	0.09	0.16
OBD_4	1.16	4000	2.61	2.64
OBD_8	0.51	8000	2.95	2.92
OBD_{16}	1.44	16000	0.28	0.52
OBD_{32}	1.08	32000	0.51	0.24
OBD_{46}	1.66	46000	0.72	0.95

Benchmark $A_x \mid A \in \{DES, FFT, HIS, IMF, OBD\}$: benchmark of application A deployed with x cores
RSD_{step}: standard deviation of observed throughput over the last 5 steps while determining the number of messages
NoM: number of external input messages used for the experiment
$PD_{Throughput}$: percentage difference in throughput between two runs
$PD_{Latency}$: percentage difference in latency between two runs

Table 7.3: Set up for experiment CS-dbp vs CS-pbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-dbp

In this experiment, all applications are set up with a high level of concurrency, i.e. having a very large number of sub-network instances. This creates a large number of tasks, i.e. increases the conductor's workload and therefore highlights its efficiency which is the main implementation difference between two schedulers. In addition, E_{sc} is set as one RC invocation so that the priority function is evaluated frequently and that helps to contrast the performance of these two schedulers.

All applications were executed on a shared memory machine with 4 AMD Opteron™ 6174 12-core Processors and 256GB of shared memory. Of the total 48 cores, 2 are used to imitate the source producing external input messages and the sink consuming external output messages.

All applications are evaluated in terms of peak throughput and processing latency when the peak throughput is achieved. To achieve peak throughput, the source is implemented with a greedy manner, i.e. the source generates external input messages as much as the application can consume. The number of external input messages is experimentally determined. It is initially set as an initial value iv and gradually increased for multiple

Benchmark	RSD_{step}	Number of Messages	$PD_{Throughput}$	$PD_{Latency}$
DES_2	0.02	2000	3.38	3.57
DES_4	0.04	4000	0.09	0.04
DES_8	0.03	8000	0.06	0.19
DES_{16}	0.03	16000	0.01	0.97
DES_{32}	0.02	32000	0.03	0.25
DES_{46}	0.03	46000	0.02	0.26
FFT_2	0.08	3600	0.03	0.02
FFT_4	0.05	7200	0.15	0.21
FFT_8	0.03	14400	0.41	0.48
FFT_{16}	0.02	28800	0.08	0.19
FFT_{32}	0.02	57600	0.06	0.11
FFT_{46}	0.04	82800	0.66	1.3
$HIST_2$	0.06	2400	0.19	0.19
$HIST_4$	0.06	4800	0.25	0.42
$HIST_8$	0.08	9600	0.89	0.94
$HIST_{16}$	0.11	19200	3.29	3.02
$HIST_{32}$	0.03	38400	0.61	0.55
$HIST_{46}$	0.05	55200	0.19	0.27
IMF_2	0.02	2000	0.37	0.37
IMF_4	0.09	4000	0.22	0.07
IMF_8	0.07	8000	0.59	0.42
IMF_{16}	0.12	16000	1.71	1.86
IMF_{32}	0.07	32000	3.52	3.2
IMF_{46}	0.04	46000	2.36	2.2
OBD_2	0.08	2000	0.61	0.61
OBD_4	0.37	4000	1.73	1.65
OBD_8	0.16	8000	2.35	2.25
OBD_{16}	0.32	16000	0.3	0.18
OBD_{32}	0.17	32000	0.72	0.48
OBD_{46}	1.03	46000	0.32	0.15

Benchmark $A_x \mid A \in \{DES, FFT, HIS, IMF, OBD\}$: benchmark of application A deployed with x cores
RSD_{step}: standard deviation of observed throughput over the last 5 steps while determining the number of messages
NoM: number of external input messages used for the experiment
$PD_{Throughput}$: percentage difference in throughput between two runs
$PD_{Latency}$: percentage difference in latency between two runs

Table 7.4: Set up for experiment CS-dbp vs CS-pbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-pbp

steps until the observed throughput converges. In each step, the number of external input messages is increased by a value of iv. Figure 7.5 demonstrates the convergence of throughput of the IMF application on CS-dbp over 20 steps. We denote the standard deviation of observed throughput over the last 5 steps as RSD_{step}. In our experiment, the number of external input messages is increased gradually until RSD_{step} is less than 2%. The final number of external input messages is denoted as NoM. The RSD_{step} and NoM values of all benchmarks of CS-dbp and CS-pbp are shown in Table 7.3 and Table 7.4 respectively. We use $A_x \mid A \in \{DES, FFT, HIS, IMF, OBD\}$ to denote the benchmark of application A deployed with x cores.

In addition, to show the stability of the results, we performed each benchmark twice. The difference between the results of these two runs is relative small. In particular, the percentage difference in all benchmarks is less then 4%. Also, all the benchmarks are implemented in S-Net where computation nodes do not maintain persistent states and the system is free of deadlock. For this reason, the small difference between two runs shows that the system is unlikely to be trapped in atypical behaviour within a run.

Table 7.3 and Table 7.4 show the percentage difference of throughput ($PD_{Throughput}$) and percentage difference of latency ($PD_{Latency}$) between two runs in CS-dbp and CS-pbp.

7.4.3.2 Experiment Result

Figure 7.6 shows the comparison between two schedulers CS-dbp and CS-pbp. Neither scheduler provides superior performance compared to the other. There is almost no difference in throughput of DES. For other applications, CS-dbp is better for some cases and CS-pbp is better for others. For example, throughput of HIST in CS-dbp is respectively 1.2%, 1.9%, 6.7%, 8.5%, 5.5% and 8.2% better on 2, 4, 8, 16, 32 and 46 cores. However, the throughput of FFT in CS-pbp is 17.4%, 3.5%, and 1.8% better on 16, 32 and 46 cores respectively. Similarly, CS-pbp brings better latency for some cases and worse latency for others. The best case for CS-pbp is FFT running on 16 cores where the latency is 26% better and the worst case is OBD on 8 cores where the latency is 17.6% worse.

The scalability of both schedulers appears to be good for all applications except for FFT. In the FFT implementation, there are $\star$-combinators which create multiple serial replicators. These tasks have very short RC invocations and therefore it causes more task requests to the conductor. That potentially increases the conductor's response time and therefore causes workers to wait. On 46 cores, with its large number of $\star$-combinators, FFT has not scaled well for both schedulers.

7.5 Evaluation: CS-dbp vs Default LPEL Scheduler

In this section, we compare the performance of these two scheduler against the default LPEL scheduler. As shown in the previous section, two proposed schedulers CS-dbp and CS-pbp have similar performance. We therefore choose CS-dbp as the representative. The default LPEL scheduler is denoted as DS. The comparison is shown in terms of the peak throughput and the corresponding processing latency when the peak throughput is achieved.

7.5.1 Experimental Set Up

The experiment is set up in a similar way to the experiment in Section 7.4. It uses the same set of applications and platform. Also, 2 cores of the platform are used to imitate the source producing external input messages and the sink consuming external output messages.

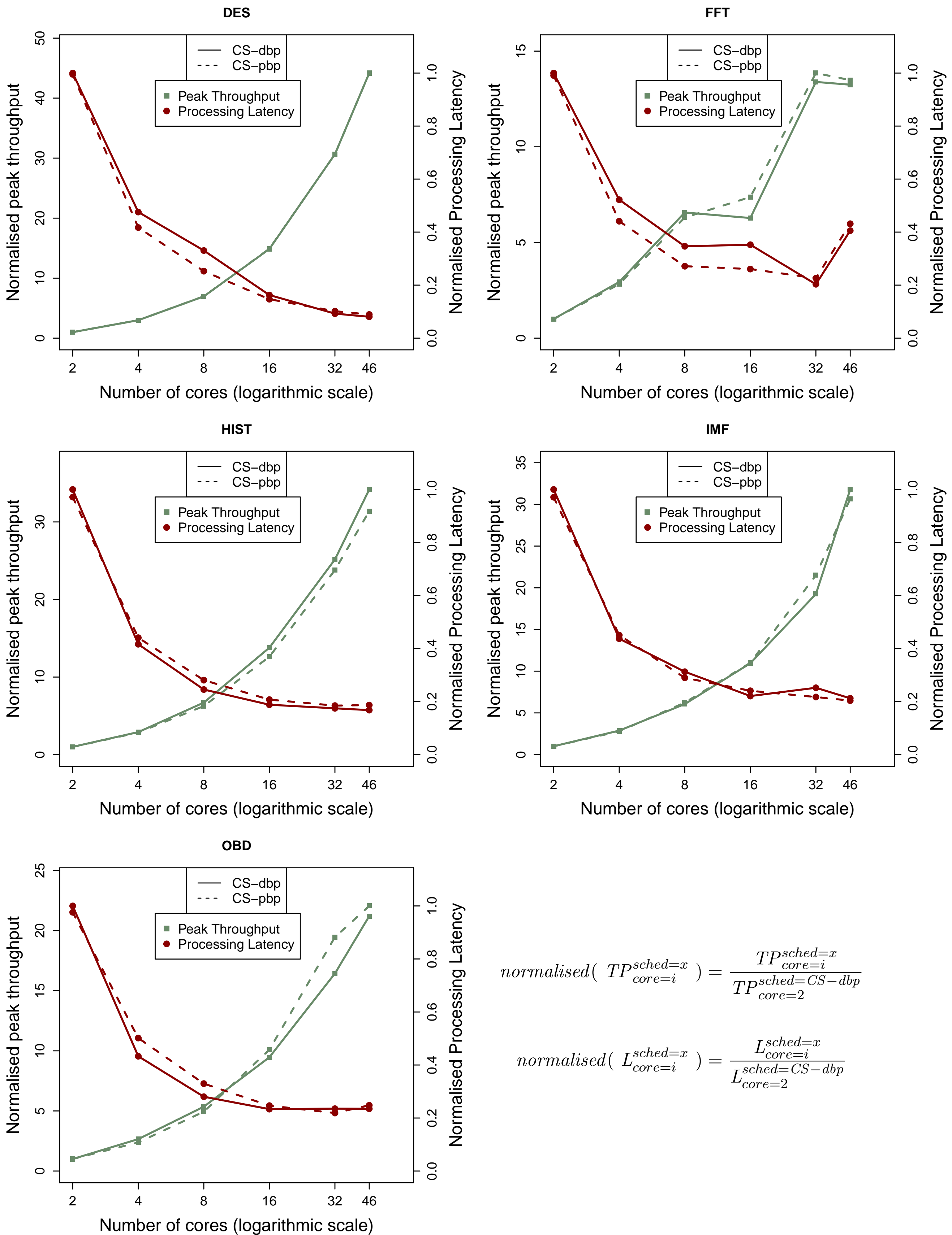


FIGURE 7.6: Normalised throughput and latency of CS-dbp and CS-pbp on various applications

Benchmark	RSD_{step}	Number of Messages	$PD_{Throughput}$	$PD_{Latency}$
DES_1	1.35	500	0.04	0.02
DES_2	0.89	1000	0.01	0.03
DES_4	0.53	2000	0.01	0.05
DES_8	0.28	4000	0.11	0.13
DES_{16}	0.34	8000	0.32	0.38
DES_{32}	0.43	16000	0.04	0.03
DES_{46}	0.21	23000	0.02	0.02
FFT_1	0.10	900	0.33	0.31
FFT_2	0.68	1800	0.02	0.08
FFT_4	0.45	3600	0.27	0.18
FFT_8	0.33	7200	1.12	1.19
FFT_{16}	1.49	14400	0.58	0.55
FFT_{32}	0.14	28800	0.51	0.42
FFT_{46}	0.07	41400	3.89	3.46
$HIST_1$	0.57	600	0.06	0.09
$HIST_2$	0.50	1200	0.63	1.1
$HIST_4$	0.47	2400	0.27	0.71
$HIST_8$	0.11	4800	0.13	0.12
$HIST_{16}$	0.05	9600	0.18	0.14
$HIST_{32}$	0.13	19200	0.83	0.67
$HIST_{46}$	0.08	27600	0.24	0.22
IMF_1	0.36	500	0.08	0.06
IMF_2	0.73	1000	0.28	0.24
IMF_4	1.17	2000	0.11	0.14
IMF_8	0.96	4000	0.34	0.28
IMF_{16}	0.29	8000	0.33	0.42
IMF_{32}	0.21	16000	0.0	0.84
IMF_{46}	0.05	23000	0.65	0.83
OBD_1	0.11	500	0.01	0.03
OBD_2	0.30	1000	0.14	0.21
OBD_4	0.09	2000	0.21	0.26
OBD_8	0.20	4000	0.1	0.23
OBD_{16}	0.73	8000	1.81	2.12
OBD_{32}	0.59	16000	0.22	0.2
OBD_{46}	0.39	23000	0.01	0.61

Benchmark $A_x \mid A \in \{DES, FFT, HIS, IMF, OBD\}$: benchmark of application A deployed with x cores
RSD_{step}: standard deviation of observed throughput over the last 5 steps while determining the number of messages
NoM: number of external input messages used for the experiment
$PD_{Throughput}$: percentage difference in throughput between two runs
$PD_{Latency}$: percentage difference in latency between two runs

TABLE 7.5: Set up for experiment DS vs CS-dbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in DS

Note that each application in this experiment is set up with an appropriate concurrency level. That means the application has a substantial number of tasks to avoid the case where workers idle because there is no available task. At the same time, the number of tasks should not too large to avoid boosting the overhead.

We derived the scheduling cycle E_{sc} by experiments on these 5 applications with different values of E_{sc} from 1 to 30 RC invocations. The observed difference in throughput and latency has been relatively small. This shows that the task-switching overhead in LPEL is negligible. Thus, for the further experiments we choose an arbitrary value for E_{sc} in the range of 1 to 30 RC invocations.

Similarly to the experiment in Section 7.4, the number of external input messages is derived experimentally by gradually incrementing them over several steps. The increment process stops when the relative standard deviation of observed throughput over

Benchmark	RSD_{step}	Number of Messages	$PD_{Throughput}$	$PD_{Latency}$
DES_2	0.21	1000	0.06	0.07
DES_4	0.24	2000	0.11	0.02
DES_8	0.24	4000	0.07	0.42
DES_{16}	0.28	8000	0.04	0.23
DES_{32}	0.22	16000	0.0	1.5
DES_{46}	0.26	23000	0.09	0.97
FFT_2	0.93	1800	0.58	0.58
FFT_4	0.19	3600	0.15	0.22
FFT_8	0.11	7200	0.01	0.12
FFT_{16}	0.11	14400	0.02	0.42
FFT_{32}	0.42	28800	0.03	0.12
FFT_{46}	0.38	41400	0.05	0.09
$HIST_2$	0.32	1200	0.09	0.08
$HIST_4$	0.12	2400	0.08	0.22
$HIST_8$	0.07	4800	0.04	0.11
$HIST_{16}$	0.17	9600	0.16	0.58
$HIST_{32}$	0.17	19200	0.87	0.74
$HIST_{46}$	0.26	27600	0.3	0.1
IMF_2	0.22	1000	0.11	0.12
IMF_4	0.21	2000	0.32	0.59
IMF_8	0.16	4000	0.07	0.15
IMF_{16}	0.26	8000	0.05	1.01
IMF_{32}	0.05	16000	0.26	0.1
IMF_{46}	0.26	23000	0.33	0.32
OBD_2	0.07	1000	0.04	0.04
OBD_4	0.02	2000	0.0	0.12
OBD_8	0.53	4000	0.11	0.13
OBD_{16}	0.93	8000	0.27	0.06
OBD_{32}	0.47	16000	0.82	0.73
OBD_{46}	0.47	23000	0.19	1.33

Benchmark $A_x \mid A \in \{DES, FFT, HIS, IMF, OBD\}$: benchmark of application A deployed with x cores
RSD_{step}: standard deviation of observed throughput over the last 5 steps while determining the number of messages
NoM: number of external input messages used for the experiment
$PD_{Throughput}$: percentage difference in throughput between two runs
$PD_{Latency}$: percentage difference in latency between two runs

Table 7.6: Set up for experiment DS vs CS-dbp: RSD_{step}, NoM, $PD_{Throughput}$, and $PD_{Latency}$ of all benchmarks in CS-dpb

the last five steps (RSD_{step}) is less than 2%. The RSD_{step} and number of external input messages (NoM) are shown in Table 7.5 for DS and in Table 7.6 for CS-dpb. Also, each benchmark is performed twice showing that the difference in results between two runs is small. In particular, the difference in all benchmarks is less than 4%. The percentage difference of throughput ($PD_{Throughput}$) and percentage difference of latency ($PD_{Latency}$) between two runs are shown in Table 7.5 for DS and in Table 7.6 for CS-dpb.

7.5.2 Performance Comparison

Figure 7.7 demonstrates the comparison in performance and throughput scalability between CS-dbp and DS. We dedicate one worker as the conductor and we only measure CS-dbp with 2 or more cores. Since the relative overhead time of CS-dbp is one CPU core for the conductor, the peak throughput is better in DS when the number of CPU cores is small. When the number of cores increases, this overhead is reduced and the peak throughput of CS-dbp is improved. In the case of 46 cores, the peak throughput of

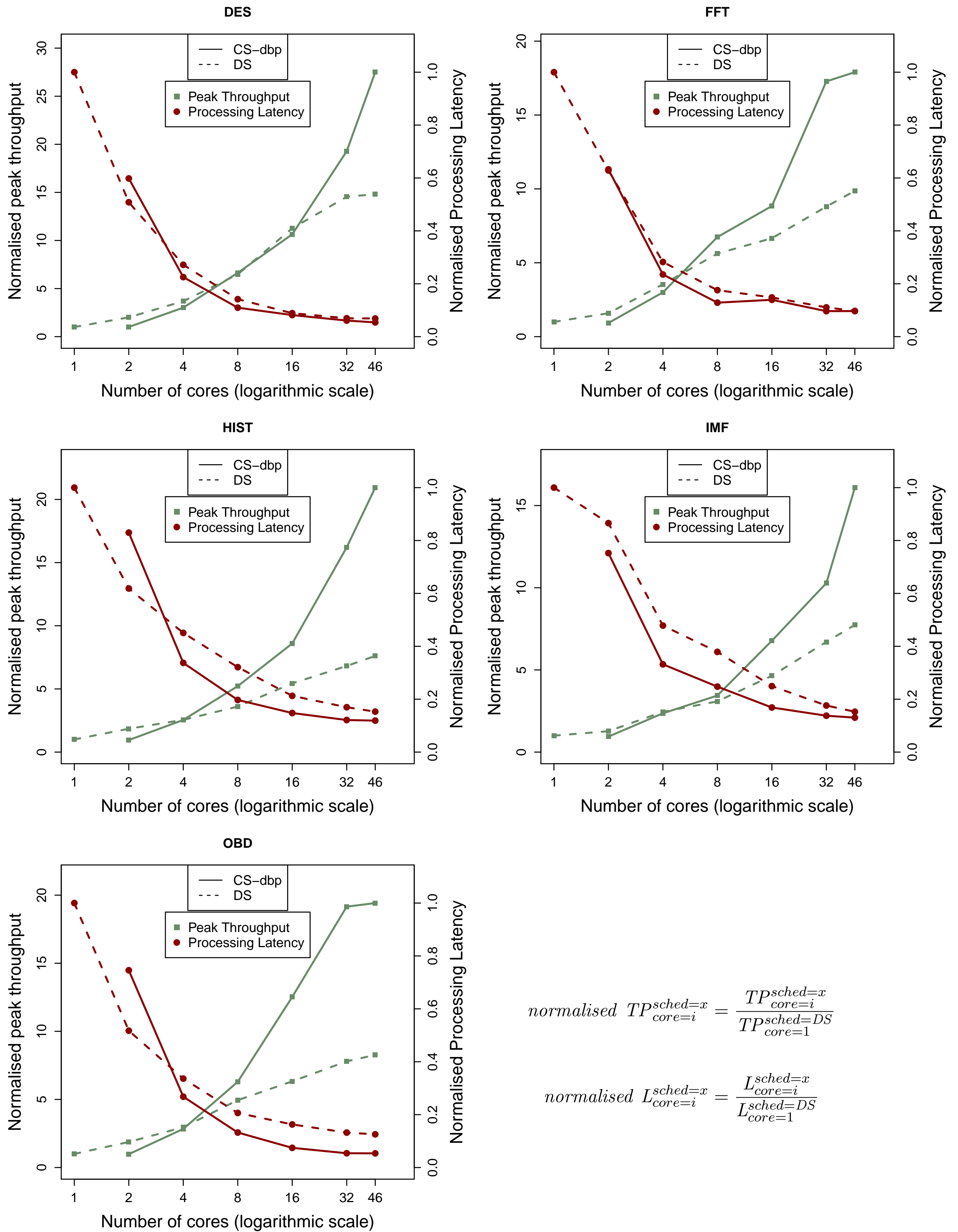


FIGURE 7.7: Normalised peak throughput and processing latency (with $\lambda = TP_{peak}$) of CS-dbp and DS on various applications

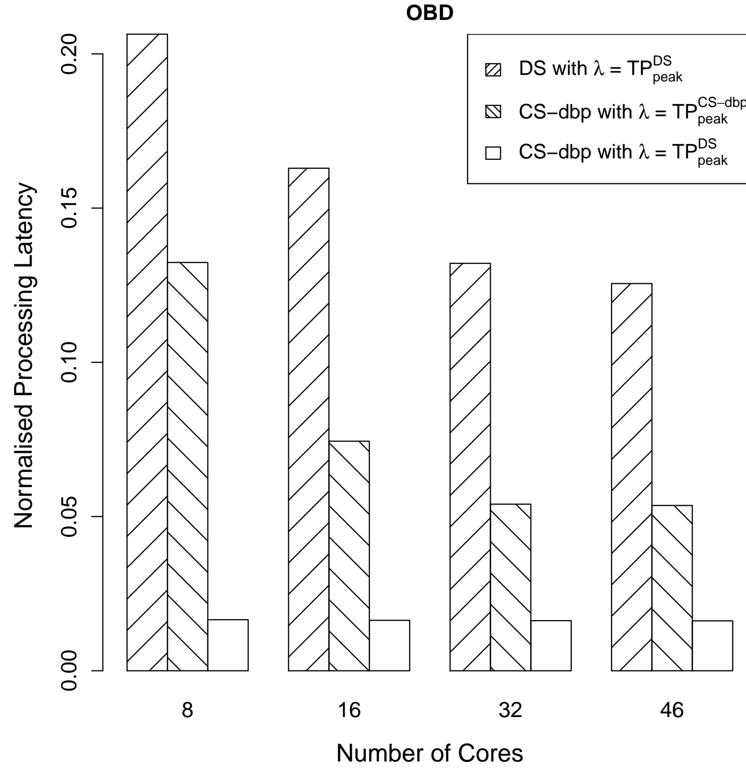


FIGURE 7.8: Processing latency comparison with different arrival rates

CS-dbp is significantly higher than DS. In particular, the peak throughput of the DES, FFT, HIST, IMF and OBD are respectively 1.8, 1.8, 2.7, 2.0, and 2.3 times higher with CS-dbp.

The processing latency of CS-dbp on 2 cores is better than DS for IMF and the same for FFT despite the higher overhead of CS-dbp. Starting from 4 cores, the processing latency of CS-dbp is better than or equal to DS for all applications. Note that the processing latency is measured when the peak throughput is achieved. CS-dbp provides higher peak throughput than DS in most of the cases, i.e. the applications can cope with a higher arrival rate. Furthermore, if the arrival rate for CS-dbp gets reduced down to the peak throughput of DS, then CS-dbp will exercise a significantly lower processing latency. Figure 7.8 demonstrates this for the OBD application. With $\lambda = TP_{peak}$, i.e. when two schedulers are compared with their own peak throughput, the processing latency of CS-dbp is 1.5 to 2.4 times lower than DS. With $\lambda = TP^{DS}_{peak}$, i.e. when two schedulers are compared with the same arrival rate, the processing latency of CS-dbp is 7.7 to 12.4 times lower than DS. The cases of 2 and 4 cores are not shown because the peak throughput of CS-dbp is smaller or equivalent to DS, as mentioned above.

7.5.3 Scalability Comparison

The processing latency depends on the concurrency level of the RPS which is reflected in the structure of the program. For this reason, the comparison in the latency scalability between two schedulers is not so appropriate. We therefore focus on the throughput scalability. The results in Figure 7.7 show that CS-dbp has a better throughput scalability than DS.

For the DES application, with 2 to 16 cores CS-dbp and DS scale at the same rate. DES is a special application where the RPS structure consist of multiple pipelines. Each pipeline has 16 tasks with the same amount of computations. As DS uses a round-robin approach to map tasks to cores, it creates a load balanced mapping when the number of tasks is a multiple of the number of cores. In this case, the relative idling time is minimal and therefore the best throughput is achieved. With 32 and especially 46 cores, the number of tasks is not a multiple of the number of cores, the round-robin mapper of DS does not provide a good load balance. Consequently the throughput does not scale well for DS. In contrast, the scalability of CS-dbp is not affected and overtakes DS.

7.6 Evaluation: CS-dbp vs Centralised Scheduler with Random Priority

We evaluate the heuristic priority function of CS-dbp by comparing it with the random task priority. In the CS-dbp scheduler, the priority of a task varies during its lifetime. To access the demand-based priority, we use the centralised scheduler that dynamically assigns a random priority for *ready* tasks before they are added to the CTQ. This scheduler is denoted as CS-drp. The experiment uses the same set up as in Section 7.5.

Since the behaviour of all applications are quite similar, we present here one illustrated case of the HIST application in Figure 7.9. As explained in Section 7.2, with a bounded $\overline{M_{cp}}$ the throughput is maximised when the relative idling time and the relative overhead time are minimised. As $\overline{M_{cp}}$ is controlled by entry and exit tasks, the random task priority does guarantee the $\overline{M_{cp}}$ to be bounded although the chance of unbounded growth is low. Using the centralised approach, the CS-drp has minimised the relative idling time $\widetilde{W}$. CS-drp has less time overhead $\widetilde{O}$ than CS-dbp because it does not need to monitor the stream fill level and keep track of the task-stream relationship. Therefore when $\overline{M_{cp}}$ is bounded the peak throughput of CS-drp is better than CS-dbp.

Since the stream structure of the program is cloned into more copies for more cores, the number of tasks and streams is increased according to the number of cores. The

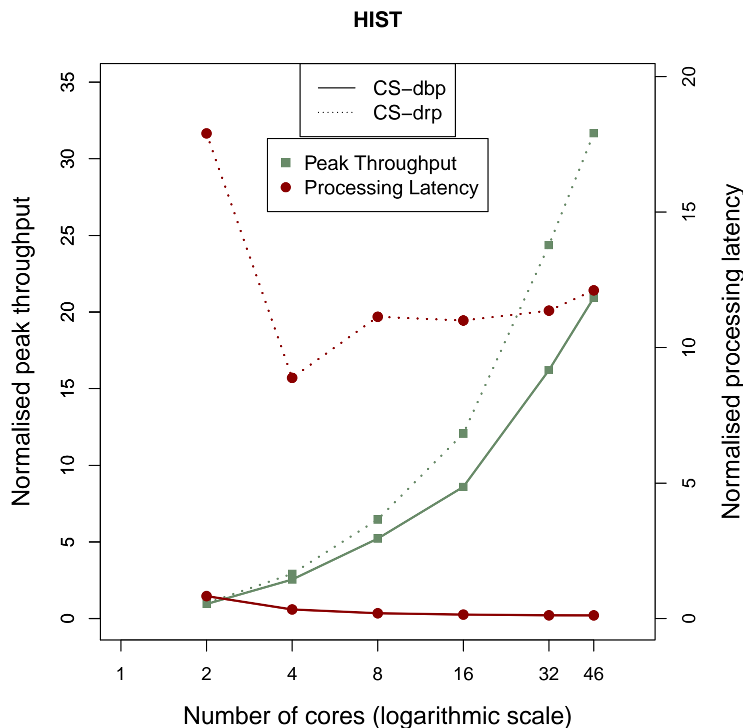


Figure 7.9: Performance of HIST using CS-dbp and CS-drp

overhead for monitoring tasks and streams in CS-dbp increases when the the number of cores increases. The difference in throughput between CS-dbp and CS-drp is higher for higher numbers of cores.

In contrast, the processing latency of CS-dbp is significantly better than CS-drp for all numbers of cores. This shows that the proposed priority function has a meaningful influence on the processing latency. However, the overhead of calculating the priority function at the same time reduces the maximum throughput.

7.7 Chapter Summary

This chapter considered various properties of stream programs to provide efficient heuristics to obtain good load balancing with online scheduling. This chapter introduced two heuristic approaches: demand-based priority (CS-dbp) and position-based priority (CS-pbp). CS-dbp uses the state of the input and output streams of each node to define its priority. As the stream state is dynamic, CP-dbp requires a perpetual re-evaluation. CS-pbp uses the notion of position which needs to be evaluated only once, when the respective node is created. Although the CS-pbp heuristic avoids the perpetual re-evaluation of CS-dbp, our experimental evaluation indicates that the throughput and

latency achieved by CS-pbp is just comparable to that of CS-dbp. This is because in our implementation of CS-pdp priorities are vectors that come with a significantly more expensive comparison operation, whereas CS-pdp priorities are numeric scalars that can be compared efficiently.

Furthermore, this chapter compared CS-dbp with the default scheduler of LPEL which does not not deploy knowledge about the structure and state of the RSP. The experimental results show that CS-dbp offers significant improvements of throughput compared to the default scheduler. For 46 cores the throughput showed improvements by a factor of 1.6 to 2.7. When limiting the arrival rate of the new scheduler down to the peak throughput of the default scheduler, we observed at the same time improvements of the latency by a factor of 7.7 to 12.4 for the OBD application on 8 or more cores.

Chapter 8

Mapping Reactive Stream Programs onto Distributed Systems

In this chapter, we propose a graph partitioning method particularly suitable for optimising the throughput of RSPs on heterogeneous distributed platforms. As an old NP-hard problem [GJS76], graph partitioning has a significant volume of existing work. The usual solutions are heuristic and approximation algorithms trying to divide a graph into separated partitions to optimise an objective. While the classical and well investigated objective is to equalise the size of partitions first and then minimise the total cuts between them, this objective is not necessarily sufficient for all problems. For RSPs, the throughput is decided not purely by the workload on each partition, but also by the communication cost between each pair of partitions. It is even more complicated when the distributed platform has heterogeneous resources and communication bandwidth is not uniform among them.

This chapter introduces two new heuristics to capture the problem space of graph partitioning for RSPs to optimise throughput. Although the thesis targets general RSPs (as shown in Table 2.2), these two new algorithms are suitable to a more restricted class of RSPs with relatively stable node behaviour and relatively static program structures. For RSPs with highly variable node computational behaviour and highly dynamic program structures, the proposed algorithms can be used for initial partitioning. During runtime when these properties vary, an adaptation method is required to repartition the RSP.

8.1 Mapping RSPs onto Distributed Platforms by Graph Partitioning

This section explains the usage of graph partitioning methods for mapping RSPs onto distributed platforms. The problem of mapping an RSP onto a distributed platform consisting of multiple PEs can be viewed as the partitioning and mapping of the task graph of the RSP to all the PEs.

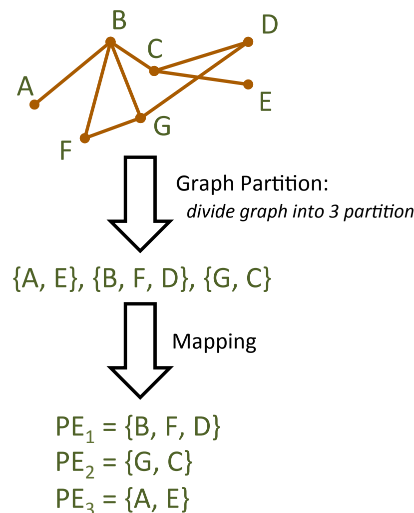


Figure 8.1: Graph partitioning and mapping in two phases

One technique is to use a graph partitioning algorithm to divide the task graph into as many partitions as the number of PEs, and then apply a mapping algorithm to assign individual partitions onto PEs. The technique is demonstrated in Figure 8.1, including two phases known as `graph partitioning` and `mapping`. The graph partitioning phase is applied first to divide the task graph into multiple partitions. The mapping phase is then applied to assign the generated partitions to PEs. This technique is often used for mapping parallel programs onto uniform distributed platforms where all the PEs are homogeneous. The graph partitioning algorithm usually aims to obtain the load balance. In the case where all PEs of the platform are homogeneously connected, the graph partitioning algorithm also needs to minimise the communication cost. The mapping in this case is a straightforward process where each partition is exclusively mapped to an arbitrary PE. In cases where the connections among PEs are not homogeneous, the mapping algorithm is designed to minimise the communication cost.

Another technique is to integrate the mapping work into the graph partitioning algorithm as shown in Figure 8.2. This technique requires us to assign each PE with an unique identification number. Let n be the number of the PEs, where each PE is enumerated from 1 to n. The graph partitioning algorithm is then used to divide the task graph into

n partitions, each of which is also assigned with a unique identification number from 1 to n. The identification number of each partition indicates the PE that the partition is mapped to, i.e. partition i is assigned to PE i with $i = 1..n$. This technique is suitable for heterogeneous distributed platforms where each PE has a different configuration since it attaches each partition to a PE before the partition is generated. This gives the graph partitioning algorithm more control to generate partitions so that the goal is optimised.

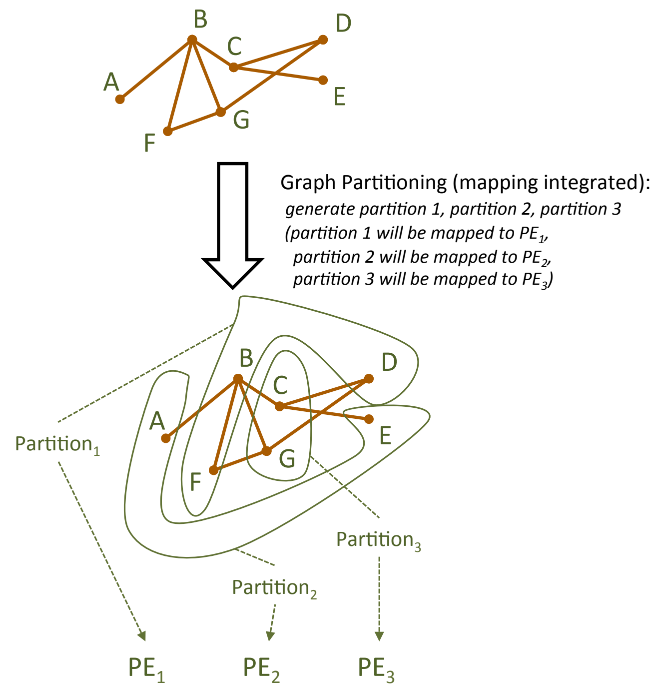


FIGURE 8.2: Graph partitioning with integrated mapping

The latter technique is chosen to develop the graph partition algorithms in this chapter as we target the mapping of RSPs onto heterogeneous distributed platforms. The mapping problem is integrated into the graph partitioning problem by coupling each partition with the PE which has the same identification number. The following section presents graph partitioning algorithms that divide the task graph into partitions so that the throughput is maximised.

8.2 Usage of Graph Partitions to Optimise Throughput of RSPs

8.2.1 Problem Statement

We introduce graph partitioning algorithms to generate a mapping configuration so that its throughput is maximised. Inputs of these algorithms include the task graph of the

RSP and the target graph of the distributed platform. The task graph is denoted as $\mathcal{G} = (\mathcal{T}, \mathcal{S})$ where $\mathcal{T}$ is the set of tasks, and $\mathcal{S}$ is the set of streams. The computation weight of each task T in $\mathcal{T}$ is defined as the average time that T requires to performs its contribution to an external input message. This amount of time depends on the PE onto which the task is mapped. The computation weight of task T on processing element R is notated as $w^R(R)$ and is measured in seconds. A stream connecting two tasks T_i and T_j is denoted as $S_{T_iT_j}$. The communication weight of the stream is denoted as $w(S_{T_iT_j})$. It equals to the average amount of data to be transferred over $S_{T_iT_j}$ for the completion of an external input message. It is the average total size of messages that the stream $S_{T_iT_j}$ transfers during its contribution to each external input message. The unit of the stream weight is Byte/message. The values of $w^R(T)$ and $w(S_{T_iT_j})$ can be easily obtained using an appropriate monitoring framework, for example the one presented in Chapter 6.

The target graph is denoted as $\mathcal{H} = (\mathcal{R}, \mathcal{L})$ where $\mathcal{R}$ is the set of vertices, also the set of PEs; and $\mathcal{L}$ is the set of edges, each of which is the communication link between PEs. Each PE, R, has weight $w(R)$, which equates to its number of cores. A communication link between two processing elements R_i and R_j is denoted as $L_{R_iR_j}$ and has weight $w(L_{R_iR_j})$ which equates to its bandwidth in Byte/s.

The graph partitioning problem involves dividing G into $|\mathcal{R}|$ partitions, each of which has a unique identification number from 1 to $|\mathcal{R}|$. The partition i is mapped to PE R_i with $i = 1..|\mathcal{R}|$. The set of partitions is called a mapping configuration $MpC = \{P_R \mid R \in \mathcal{R}\}$. The goal of our graph partitioning algorithms is to find the mapping configuration so that the throughput is maximised.

In Section 5.3.1, we presented a method to calculate the throughput of RSPs when deployed on a distributed platform with a mapping configuration. Given a mapping configuration MpC, the throughput is the minimum value of computation throughput of each partition, and the communication throughput of each pair of partitions:

$$TP(MpC) = \min_{R_i, R_j, R_k \in R} \left(TP_{comp}\left(P_{R_i}\right), TP_{comm}\left(P_{R_j}, P_{R_k}\right)\right)$$

with

$$TP_{comp}(P_R) = \frac{w(R) - \widetilde{W_R} - \widetilde{O_R}}{\overline{C_R}} = \frac{w(R) - \widetilde{W_R} - \widetilde{O_R}}{\sum\limits_{T \in P_R} w^R(T)}$$

$$TP_{comm}(P_{R_i}, P_{R_j}) = \frac{w(L_{R_iR_j})}{Comm(P_{R_i}, P_{R_j})}$$

8.2.2 Partitioning RSPs with Variable Node Computational Behaviour and Dynamic Program Structures

To generate the task graph $\mathcal{G} = (\mathcal{T}, \mathcal{S})$, we need to use the monitoring framework in Chapter 6 to extract properties of RSPs including the task weight, the stream weight and the structure of RSPs. Since the approach of using graph partitioning for mapping RSPs is an offline approach, it requires these properties to be relatively constant during runtime. Therefore, RSPs with highly varied task weights, stream weights and program structures during the runtime are not addressed by our proposed algorithms. However, the proposed algorithms can be used to find an initial mapping configuration by fixing the varied properties with average values. During runtime when these properties change, an adaptation method would be required to repartition the RSP.

8.3 New Graph Partition Algorithms to Optimise Throughput of RSPs

In this section, we focus on finding the mapping configuration so that the throughput is maximised. We assume that the local scheduler of each PE has predictable relative idling time $\widetilde{W}$ and relative overhead time $\widetilde{O}$. We also assume that these values are not affected by the partitioning method.

We introduce two partitioning algorithms to generate a mapping configuration so that its throughput is maximised. The first algorithm, called *KL-Adapted* (KLA), is a trivial adaptation from the well-known graph partitioning algorithm, Kernighan-Lin [KL70]. The second one, called *Congestion Avoidance* (CA), operates in a similar way but instead of considering all possible moves, the method detects the congestion point and examines only moves that can help to improve the congestion point.

For convenience, we denote by $par(MpC, T)$ the partition in MpC that task T belongs to.

8.3.1 KL-Adapted Algorithm

The original Kernighan-Lin algorithm (KL) [KL70] aims to divide a graph into two partitions such that they are balanced in terms of the number of vertices with a minimum number of edges across them. The algorithm introduces the gain of a vertex as the total of edge cut which will be decreased if the vertex is moved to the complimentary partition. The gain of each vertex is calculated based on internal edges connecting the vertex with

Function	Input	Output	Semantics
rndGen	Task graph $\mathcal{G}$ Target graph $\mathcal{H}$	Mapping configuration	Generate a random mapping configuration of $\mathcal{G}$ onto $\mathcal{H}$
TP	Mapping configuration MpC	Throughput	Calculate the throughput of mapping configuration MpC
append	List of elements L Element E	New list	Append E to L and return L
move	Mapping configuration MpC Task T Partition P	New mapping configuration	Given the current mapping configuration MpC, the function relocates task T to partition P and returns MpC

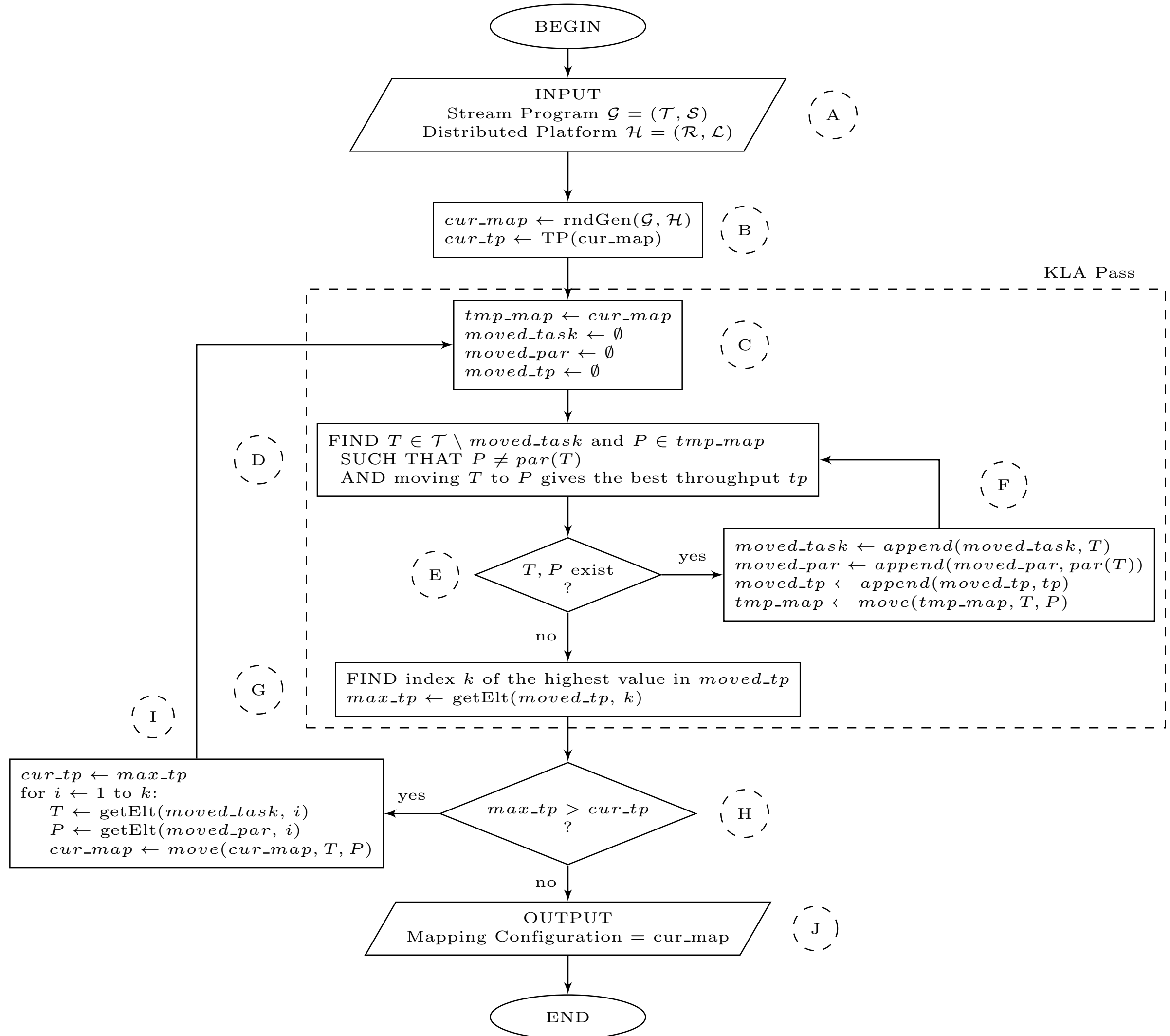
Table 8.1: Semantics of built-in functions used in KLA and CA algorithms

vertices on the same partition, and external edges connecting the vertex with vertices on the complimentary partition.

Starting with a randomly generated partition, the algorithm uses a greedy heuristic to find an optimal sequence of operations between two partitions which maximise the improvements. Each operation includes choosing a vertex from the first partition with the maximum gain to move to the second partition, and similarly choosing a vertex from the second partition with the maximum gain to move to the first partition. After each move, the gain of all vertices are updated locally by examining the moved vertex and its neighbours.

The Fiduccia and Mattheyses algorithm (FM) [FM82] is an improvement of Kernighan-Lin by using an appropriate data structure. When gain values are integers and fall in a bounded range, a set of buckets can be used to store vertices. Each bucket is labelled with a value in the gaining range. Each vertex is stored in one bucket with the label matching with the vertex's gain value. This helps to choose the best vertex to move, that is one of the vertices in the bucket with the largest label. When the gain of a vertex changes, the vertex is moved to the new bucket according to its new gain.

We adapt the Kernighan-Lin algorithm to the multiple-way partitioning approach where each operation is to move one task to a new partition. Our new algorithm is called the KL-Adapted (KLA) Partitioning Algorithm. Similar to the original Kernighan-Lin algorithm, KLA starts with a randomly generated initial mapping configuration where each task is randomly assigned to a PE. KLA then applies the greedy heuristic pass iteratively until the throughput stops increasing. We denote these passes as *KLA passes* to distinguish them from passes in the CA algorithm presented later. Each KLA pass searches for the best move operation which relocates a task to a new PE so that the throughput after the move operation is maximum. After being relocated, a task is locked so that it is moved only once during a KLA pass. This process is carried on until all tasks have been moved. At the end of the KLA pass, the sequence of move operations



The semantics of built-in functions are explained in Table 8.1

FIGURE 8.3: Flowchart of the KL-Adapted partitioning algorithm

that creates the new mapping configuration with the highest throughput are chosen to be applied.

In the original KL/FM algorithm, the gain of a vertex is tracked so that it is easy and quick to choose the best vertex to move. This is feasible since after moving a vertex, only the gain of its neighbours needs to be updated by simple arithmetic operations. In our problem, moving a vertex may change the gain values of all vertices and therefore tracking the gain value of each vertex is not beneficial. We instead keep track of the computation throughput of each partition and the communication throughput between each pair of partitions. The KLA pass in our algorithm needs to examine all possible move operations to find the best move operation. When applying a move operation, the computation and communication throughputs of involved partitions will be updated.

```
Data: 𝒢 = (𝒯, 𝒮), ℋ = (ℛ, ℒ)
Result: Map with optimal TP(Map)
cur_map ← rndGen(𝒢, ℋ) ;
cur_tp ← TP(cur_map) ;
tmp_map ← cur_map ;
cont_flag ← TRUE ;
repeat
    // KLA pass
    moved_task ← ∅ ;
    moved_par ← ∅ ;
    moved_tp ← ∅ ;
    repeat
        FIND T ∈ 𝒯 \ moved_task and P ∈ tmp_map ;
         SUCH THAT P ≠ par(T) ;
         AND moving T to P gives the best throughput tp;
        if T and P exits then
            moved_task ← append(moved_task, T) ;
            moved_par ← append(moved_par, par(T)) ;
            moved_tp ← append(moved_tp, tp) ;
            tmp_map ← move(tmp_map, T, P)
        end
    until T and P do not exist;
    // extract the best move sequence
    FIND index k of the highest value in moved_tp ;
    tmp_tp ← getElt(moved_tp, k) ;
    if tmp_tp > cur_tp then
        for i : 1 → k do
            T ← getElt(moved_task, i) ;
            P ← getElt(moved_par, i) ;
            tmp_map ← move(tmp_map, T, P) ;
        end
        cur_tp ← tmp_tp ;
        cur_map ← tmp_map ;
    else
        cont_flag ← FALSE ;
    end
until cont_flag = FALSE;
```

The semantics of built-in functions are explained in Table 8.1

Algorithm 1: Pseudo code of KL-Adapted partitioning algorithm

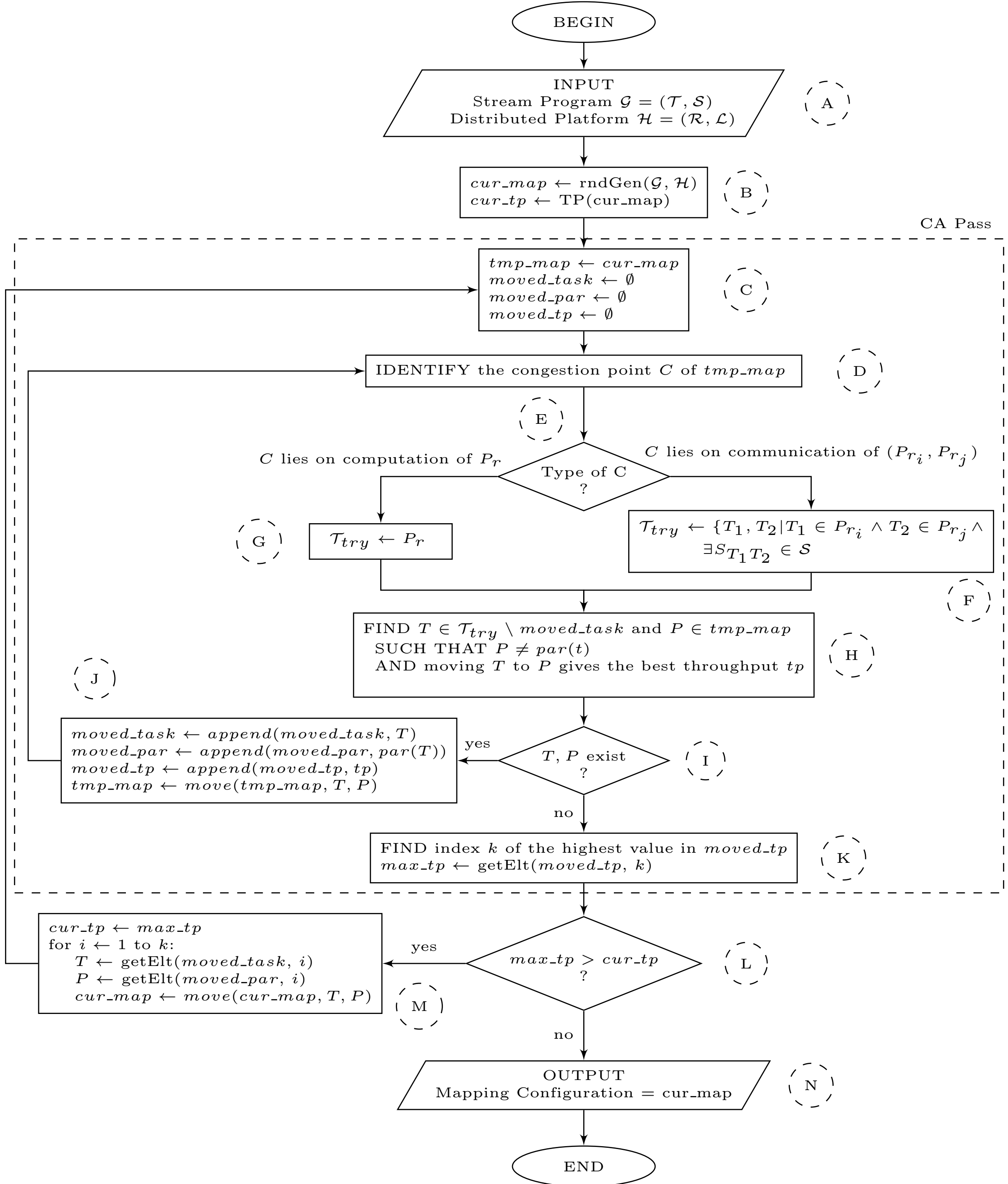
The throughput of the new mapping configuration then is calculated as the minimum value of all computation and communication throughputs.

The details of KLA are shown in Figure 8.3, labelled as steps A to J. The input to the KLA algorithm includes the RSP $\mathcal{G} = (\mathcal{T}, \mathcal{S})$, and the distributed platform $\mathcal{H} = (\mathcal{R}, \mathcal{L})$ (step A). At the beginning, a random mapping configuration is generated and its throughput is evaluated as in step B. The KLA algorithm uses three lists *moved_task*,

moved_par and *moved_tp* to store the history of move operations. Starting with the empty history, each KLA pass operates on a temporary mapping configuration *tmp_map* which is a copy of the current mapping configuration *cur_map* (step C). The KLA pass then finds the task T not in *moved_task* and the partition P so that moving T to P carries out the highest throughput compared to all other possible move operations. If the move operation exists (step E with 'yes'), it is applied to generate a new mapping configuration. The move operation is also added to the history, i.e. T is appended to *moved_task*, P is appended to *moved_par*, and the new throughput value is added to *moved_tp* (step F). Note that some move operations in the history can reduce the throughput, i.e. an element in *moved_tp* can have a smaller value than others before it. To avoid the case where a task is repeatedly exchanged between two partitions, one task is moved at most once within a pass. That means tasks in *moved_task* are not considered to move again within the current KLA pass. When all tasks have been moved once, i.e. no move operation can be found (step E with 'no'), the list *moved_tp* is examined to find the maximum throughput value *max_tp* and its index k (step G). The current KLA pass terminates and a new pass will proceed if *max_tp* is higher than the throughput of the current mapping configuration (step H with 'yes'). Before starting a new KLA pass, the current mapping configuration *cur_map* needs to be updated by applying all move operations in the history up to index k (step I). In the case where *max_tp* is not higher than the throughput of *cur_map* (step H with 'no'), that means the heuristic KLA pass can not find any better mapping configuration. The KLA algorithm therefore returns *cur_map* as the final mapping configuration (step J) and terminates.

The pseudo code of the KLA algorithm is also included in Algorithm 1. The inner loop of each KLA pass is repeated until no move is made. Since each task is moved exactly once, $|\mathcal{T}|$ moves are applied in a KLA pass, i.e. the inner loop is applied $|\mathcal{T}|$ times. For each iteration of the inner loop, the algorithm needs to search for a task T and a partition P so that relocating T to P would bring the best throughput. To find the best T and P, the algorithm needs to examine all possibilities of task relocation. The complexity of this examination is $|\mathcal{T}_{examined}| \times (|\mathcal{R}| - 1)$, where $\mathcal{T}_{examined}$ is the number of the task to be examined. For the first iteration of the inner loop, $\mathcal{T}_{examined}$ equals to $\mathcal{T}$. After each iteration, the number of the tasks to be examined is reduced by one until the last iteration where there is only one task to be examined. The complexity of the KLA pass is therefore:

$$\mathcal{O}(\sum_{i=1}^{|\mathcal{T}|} i \times (|\mathcal{R}| - 1)) = \mathcal{O}(|\mathcal{T}|^2 \times |\mathcal{R}|)$$



The semantics of built-in functions are explained in Table 8.1

FIGURE 8.4: Flowchart of the Congestion Avoidance partitioning algorithm

```
Data: G = (T, S), H = (R, L)
Result: Map with optimal TP(Map)
cur_map ← RndGen(G, H) ;
cur_tp ← TP(cur_map) ;
tmp_map ← cur_map ;
cont_flag ← TRUE ;
repeat
    // CA pass
    moved_task ← ∅ ;
    moved_par ← ∅ ;
    moved_tp ← ∅ ;
    repeat
        IDENTIFY congestion point C of tmp_map ;
        if C lies in P_i then
            T_try ← P_r ;
        end
        if C lies in (P_i, P_j) then
            T_try ← {T_1, T_2 | T_1 ∈ P_i ∧ T_2 ∈ P_j ∧ ∃S_{T_1 T_2} ∈ S} ;
        end
        FIND T ∈ T_try \ moved_task and P ∈ tmp_map ;
         SUCH THAT P ≠ par(T) ;
         AND moving T to P gives the best throughput tp ;
        if T and P exits then
            moved_task ← append(moved_task, T) ;
            moved_par ← append(moved_par, par(T)) ;
            moved_tp ← append(moved_tp, tp) ;
            tmp_map ← move(tmp_map, T, P) ;
        end
    until T does not exist;
    // extract the best move sequence
    FIND index k of the highest value in moved_tp ;
    tmp_tp ← getElt(moved_tp, k) ;
    if tmp_tp > cur_tp then
        for i : 1 → k do
            T ← getElt(moved_task, i) ;
            P ← getElt(moved_par, i) ;
            tmp_map ← move(tmp_map, T, P) ;
        end
        cur_tp ← tmp_tp ;
        cur_map ← tmp_map ;
    else
        cont_flag ← FALSE ;
    end
until cont_flag = FALSE;
```

The semantics of built-in functions are explained in Table 8.1

Algorithm 2: Pseudo Code of Congestion Avoidance partitioning algorithm

8.3.2 Congestion Avoidance Partitioning Algorithm

Similar to KLA, the Congestion Avoidance (CA) partitioning algorithm begins with an initial mapping configuration and repeats a heuristic pass until the throughput reaches an optimal value. We denote the heuristic pass here as *CA pass*. Unlike a KLA pass, a CA pass does not examine all possible move operations, but focuses on only those around the congestion point identified by inspecting the throughput formula of Equation 5.5. Within each pass, the CA identifies the congestion point of the current mapping configuration and tries move operations that potentially improve the throughput.

From Equation 5.5, the throughput of an RSP with a mapping configuration MpC is the minimum value of a set of computation and communication throughputs. A congestion point is where the throughput is settled, i.e. where the minimum value occurs. If the minimum value is $TP_{comp}(P_R)$, the congestion is said to lie on the computation of P_R. In this case, only tasks from P_R are considered to be moved to other partitions. This helps to reduce $\sum_{t \in P_R} w_R^t$ and therefore increase $TP_{comp}(P_R)$. Thus the throughput $TP(MpC)$ then improves.

Similarly if $TP_{comm}(P_{R_i}, P_{R_j})$ is the minimum value, the congestion lies on the communication between partitions P_{R_i} and P_{R_j}. In this case, only move operations that reduce the communication weight between P_{R_i} to P_{R_j} are considered. Those are move operations involving tasks which have a stream connection across P_{R_i} and P_{R_j}. Relocating these tasks potentially reduces the communication weight between P_{R_i} and P_{R_j} and therefore potentially improves $TP_{comm}(P_{R_i}, P_{R_j})$.

The details of the CA algorithm are shown in Figure 8.4, labelled as steps A to N. Taking an RSP $\mathcal{G} = (\mathcal{T}, \mathcal{S})$ and a distributed platform $\mathcal{H} = (\mathcal{R}, \mathcal{L})$ as inputs (step A), CA starts by generating a random mapping configuration *cur_map* (step B) before it starts heuristic passes. Similar to KLA, CA stores the history of move operations in three lists *moved_task*, *moved_par* and *moved_tp*. Each CA pass also starts with an empty history and a temporary mapping configuration *tmp_map* which is copied from the current mapping configuration *cur_map* (step C). By evaluating the throughput formula (Equation 5.5) on *tmp_map*, the CA pass identifies the congestion point. The type of congestion point is used to determine the set of tasks T_{try} so that reallocating them can potentially improve the congestion point (steps E, F and G). If the congestion point lies on the computation of P_R, $\mathcal{T}_{try}$ consists of tasks in P_R. If the congestion point lies on the communication of P_{R_i} and P_{R_j}, $\mathcal{T}_{try}$ includes pairs of tasks, one in P_{R_i} and one in P_{R_j}, which are connected by a stream. In step H, the algorithm examines move operations of re-allocating tasks in $\mathcal{T}_{try}$ which have not been moved during the current pass, i.e. are not in the list *moved_task*. The result of this step is the move operation

of a task T to another partition P that brings the highest throughput compared to other examined move operations. The remaining steps of the CA pass are similar to a KLA pass. The move operation is applied and added to the history if it exists (step J). Otherwise, the algorithm scans the history of move operations to find the highest throughput value max_tp and its index k (step K). The algorithm decides to update cur_map and continues a new CA pass if max_tp is better than the throughput of cur_map (step M). If not, the algorithm terminates with cur_map as the output.

The pseudo code of CA is also included in Algorithm 2. The inner loop of each CA pass is repeated until no move is made. Since each task is moved at most one, $|\mathcal{T}|$ moves are applied in a KLA pass, i.e. the inner loop is applied $|\mathcal{T}|$ times at most. For each iteration of the inner loop, the algorithm needs to search for a task T and a partition P so that relocating T to P would bring the best throughput. To find the best T and P, the algorithm needs to examine all tasks in $\mathcal{T}_{try}$. The complexity of this examination is $|\mathcal{T}_{try}| \times (|\mathcal{R}| - 1)$, where $\mathcal{T}_{try}$ is either the number of tasks in P_R if the congestion point lies on the computation of P_R, or the number of tasks in P_{R_i} and P_{R_j} if the congestion point lies on the communication link between P_{R_i} and P_{R_j}.

On average, each partition contains $\frac{|\mathcal{T}|}{|\mathcal{R}|}$ tasks. For the first iteration of the inner loop, the average value of $\mathcal{T}_{try}$ equals to $c \times \frac{|\mathcal{T}|}{|\mathcal{R}|}$, where constant $c = 1$ if the congestion point lies on the computation of a partition, or $c = 2$ if the congestion point lies on the communication links of a pair of partitions. After each iteration, the average number of the tasks to be examined is reduced by $\frac{1}{|\mathcal{R}|}$ until the last iteration the there is only one task to be examined. The complexity of the CA pass is therefore:

$$\mathcal{O}\left(\left(\frac{|\mathcal{T}|}{|\mathcal{R}|} + \frac{|\mathcal{T}|-1}{|\mathcal{R}|} + ... + \frac{|\mathcal{R}|}{|\mathcal{R}|}\right) \times (|\mathcal{R}| - 1)\right) = \mathcal{O}\left(\frac{|\mathcal{R}|-1}{|\mathcal{R}|} \times \sum_{i=|\mathcal{R}|}^{|\mathcal{T}|} i\right) = \mathcal{O}(\mathcal{T}^2 - \mathcal{R}^2)$$

8.3.3 Local Optima in Heuristic Search

Since the proposed algorithms KLA and CA are local search heuristics, they can be trapped in local optimum. There are a number of approaches to overcome this problem. One simple approach is to run the partitioning algorithm multiple times with different initial mapping configurations. Another more complicated approach is to integrate the local search heuristic into multilevel schemes. This has been shown have been shown to be successful in overcoming the localized nature of KL/FM [KK98], [HL95b]. In this work, we focus on the effectiveness of our new local search heuristic. We therefore choose to perform our algorithms multiple times with different initial mapping configurations. The approach of integrating into multilevel schemes will be considered in our future work.

8.4 Evaluation of the Partitioning Algorithms

In this section we evaluate the performance and efficiency of the two heuristic algorithms KLA and CA presented in Section 8.2. To the best of our knowledge, we are the first to use the throughput calculation using Equation 5.5 as the optimisation metric for graph partitioning. Despite using another version of this formula, the approach in [CRA09] does not consider congestion points lying on the communications but attempts to eliminate only those lying on the computations. We therefore compare our proposed heuristics with Simulated Annealing (SA) as it is a generic technique for finding the global optima of a specific function.

8.4.1 Experimental Setup

We perform our experiments with RSPs implemented by S-Net. We use the monitoring framework presented in Chapter 6 to obtained the profiling information for the RSP graph. Each PE on the distributed system has its own LPEL execution layer facilitated with the centralised scheduler CS-dbp as presented in Chapter 7. With this scheduler, the relative overhead time $\widetilde{W}$ is predictable. It is equal to one CPU core which is used as the conductor to manage the central task queue.

We use five different applications:

- DES: performs DES encryption
- OBD: detects four different types of objects from images
- HIST: calculates histograms of images
- MTI: detects moving objects on the ground from an aircraft [PHG+10]
- *S500*: a synthetic application

The first 4 applications are implemented using S-Net. Each of them contains a main sub-network that performs the application's main function. To increase the level of concurrency, parallel replication is used to create multiple instances of the main sub-network. The number of instances is decided by the number of machines of the deployed targets. Therefore, each application when deployed on different targets will have a different number of nodes and streams. The details of these applications are shown in Chapter 4.

To diversify the experiment, we include a synthetic application called *S500*. As the name suggests, it has 500 nodes. Each pair of nodes is connected with a probability of

Application	# Vertices/Nodes	# Edges/Streams
DES_4	74	76
OBD_4	98	144
$HIST_4$	122	180
MTI_4	126	140
DES_8	146	152
OBD_8	194	288
$HIST_8$	242	360
MTI_8	250	280
DES_{16}, DES_{S16}	290	304
OBD_{16}, OBD_{S16}	386	576
$HIST_{16}, HIST_{S16}$	482	720
MTI_{16}, MTI_{S16}	498	560
$S500_{16}, S500_{S16}$	500	62414

TABLE 8.2: Number of vertices and edges of evaluation benchmarks

50%. The weight of each node is randomly chosen between the minimum and maximum weights of nodes in other applications.

The target platforms of our experiments are clusters of 4, 8, and 16 machines. Each machine has 2 sockets with Xeons E5520 CPUs. Each machine also has 24 GB of shared memory. The machines are connected via a 4xDDR Infiniband network where the traffic between pairs of machines is guaranteed for a full bandwidth of 16Gbits/s. For convenience, $TARGET_i$ is used to denote the target with i machines. Note that although each machine has 8 cores, only 7 cores are used for the computation as one core is used as the conductor. Also, for each target, one of its machines needs to reserve 2 cores to simulate the source and sink of the application. The source is a process that continuously sends external input messages to the application while the sink continuously consumes its external output messages. We also include one *synthetic target* with 16 machines. The number of cores on each machine is chosen randomly from 1 to 7 cores. The bandwidth of their connections is assigned arbitrarily from 0 to 16 Gbits/s. This synthetic target is denoted as $TARGET_{S16}$.

We use $A_i \mid A \in \{DES, HIS, OBD, MTI, S500\}$ to denote for the benchmark of application A deployed on target $TARGET_i$.

Table 8.2 shows the number of vertices and edges for all benchmarks. As described above, each application when deployed on different platforms will have different numbers of nodes and streams. The more machines that the platform has, the more nodes and streams that the application includes. Note that the number of nodes is the number of vertices of the input graph. Similarly, the number of streams is the number of edges. The synthetic benchmarks $S500_{16}$ and $S500_{S16}$ come with the largest graphs. They have similar numbers of vertices as MTI_{16} and MTI_{S16} but significantly more edges.

8.4.2 Convergence Speed of KLA and CA

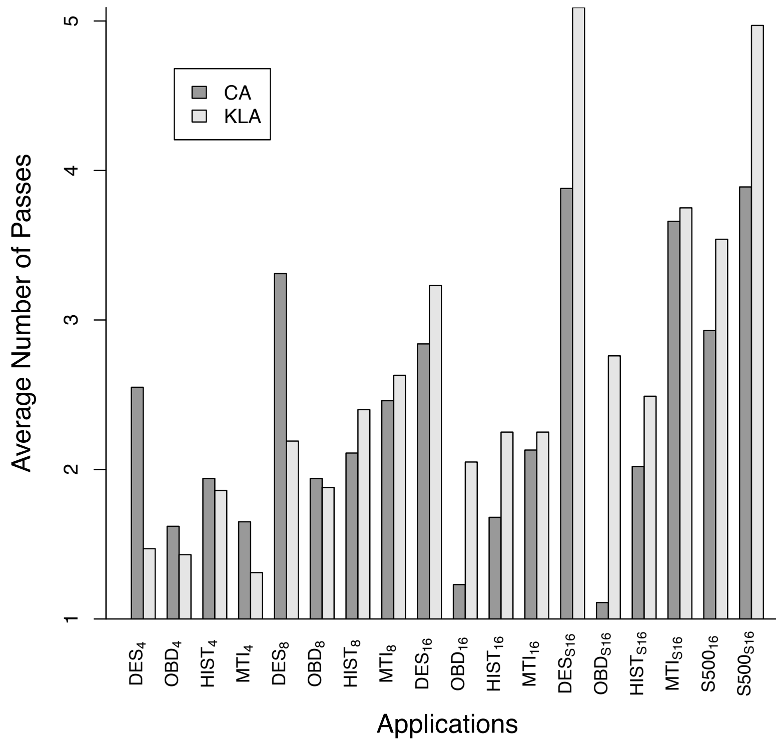


FIGURE 8.5: Comparison of KLA and CA in the average number of passes

In this section we investigate the convergence in terms of the number of passes and the execution time of two proposed algorithms, KLA and CA. Figure 8.5 shows the average number of passes of these two algorithms. The average value is calculated based on 100 runs. For all the evaluation benchmarks, both of the algorithms require a small number of passes with the highest value being 5. Comparing between KLA and CA, it is not clear which one is better in terms of number of passes. However, the difference is not significant, it is less than 1 for most of our benchmarks.

As presented in Section 8.2, each CA pass considers only move operations around the congestion point while KLA examines all possible move operations. The execution time of CA is therefore significantly less than KLA although they take similar numbers of passes. This is confirmed by the execution time ratio of KLA to CA shown in Figure 8.6. In general, KLA is significant slower than CA and it seems to be the trend that this ratio increases for benchmarks with larger numbers of vertices and edges. For example, KLA is 3 to 6 times slower for small benchmarks like DES_4, OBD_4, $HIST_4$ and MTI_4. KLA is 14 to 22 times slower for DES_8, OBD_8, $HIST_8$ and MTI_8. For very large benchmarks DES_{16}, OBD_{16}, $HIST_{16}$ and MTI_{16}, KLA takes 62 to 400 times longer to execute than CA.

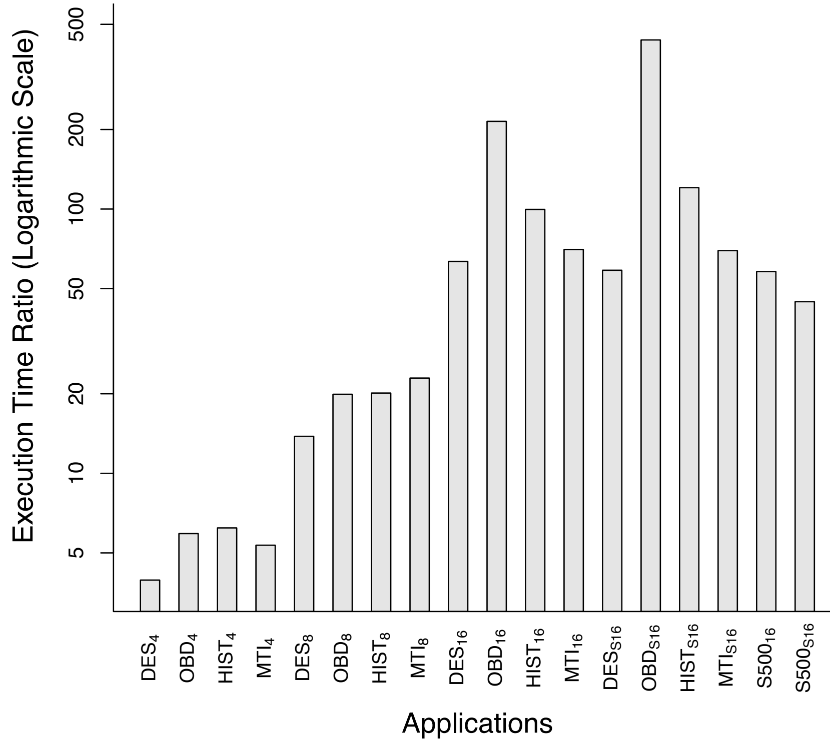


FIGURE 8.6: Execution time ratio: et_{KLA}/et_{CA}

8.4.3 Comparison with Simulated Annealing

As mentioned in Section 8.2, both CA and KLA are local search heuristics and therefore can be trapped in local optima. To overcome this, the most common strategy is to repeat the algorithms multiple times with different initial mapping configurations. We compare 50 runs of our heuristic algorithms with the generic global search algorithm, Simulated Annealing (SA) [KGV83].

Simulated annealing (SA) is a generic probabilistic metaheuristic for finding the global optimization of a given function with a large search space. SA is often more efficient than an exhaustive search for finding the nearly optimal point. Inspired by the cooling process of heated metals, the SA algorithm starts with an initial state and initial temperature T_{init} and applies an iteration of cooling steps. At each step, SA considers some neighbour state s' of the current state s. If the s' is better than s then SA moves to s', otherwise SA also moves to s' with a probability. At the end of each step, the temperature is reduced by a factor called the cooling ratio r_t. The cooling step is repeated until the temperature reaches the cooling temperature T_{min}.

The SA algorithm is used here to find the mapping configuration of an RSP so that the throughput is maximised. The psuedo code of SA is shown in Algorithm 3. Starting with a randomly generated mapping configuration, and an initial temperature, SA then applies an iteration of cooling steps. At each step, SA considers L neighbour mapping

```
Data: G = (T, S), H = (R, L)
Result: Map with optimal TP(Map)
cur_map ← RndGen(G, H) ;
cur_temp ← init_temperature ;
L ← temperature_length ;
r_t ← cooling_ratio ;
k ← Boltzmann_constant;
repeat
    // SA cooling step
    for i = 1 to L do
        T ← RndTask(T) ;
        P ← RndPar(R) so that T ≠ Par(T) ;
        new_map ← move(cur_map, T, P) ;
        δ ← TP(new_map) − TP(cur_map) ;
        if δ > 0 then
            cur_map ← new_map ;
        end
        if e^(−δ/(k×cur_temp)) > random[0, 1) then
            cur_map ← new_map ;
        else
        end
    end
    cur_temp ← r_t × cur_temp ;
until cur_temp > T_min;
```

Algorithm 3: Pseudo code of Simulated Annealing algorithm

configurations where L is the temperature length. Each neighbour mapping configuration is generated by moving an arbitrary task to an arbitrary partition. A neighbour mapping configuration is accepted if it gives a better throughput. Otherwise, it is still accepted with a probability $p = e^{\frac{-\delta}{(k \times cur_temp)}}$, where δ is the throughput different between the neighbour mapping configuration and the current mapping configuration. At the end of a cooling step, the temperature is reduced by the cooling ratio r_t. The cooling process is repeated until the temperature reaches the minimum temperature T_{min}.

$T_{init}, T_{min}, r_t, L$ are the parameters of the SA algorithm. These parameters are chosen so that acceptable results can be reached within a feasible time. By trying multiple combinations of the parameters, we found the following parameters where SA behaves well for all non-synthetic benchmarks:

- initial temperature $T_{init} = 109.00$
- minimum/cooling temperature $T_{min} = 0.02$
- cooling ratio $r_t = 0.985$
- temperature length $L = 0.1 \times neighbour_size$

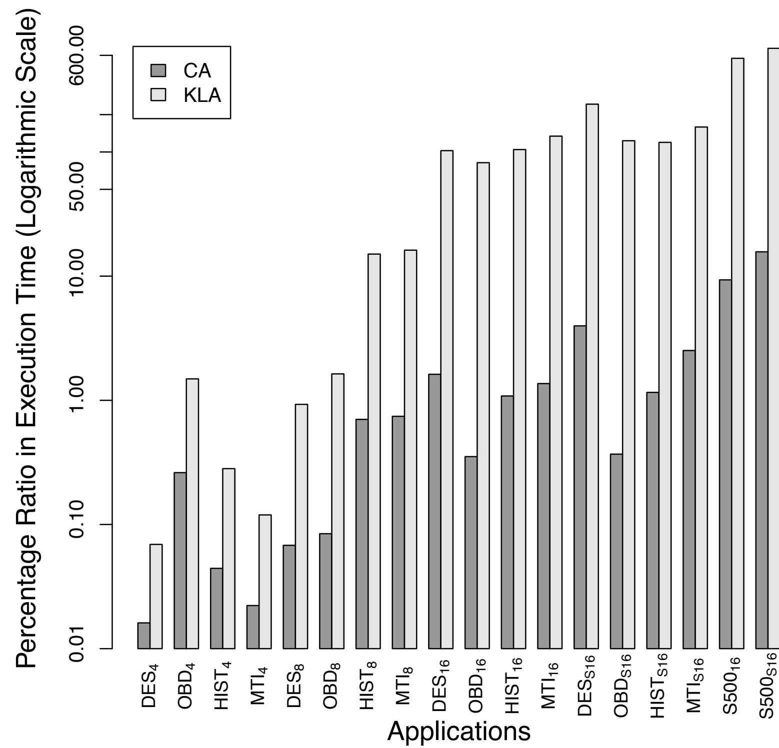


FIGURE 8.7: Execution time ratio of 50-run CA and KLA to SA ($et_{50\times CA}/et_{SA}$; $et_{50\times KLA}/et_{SA}$)

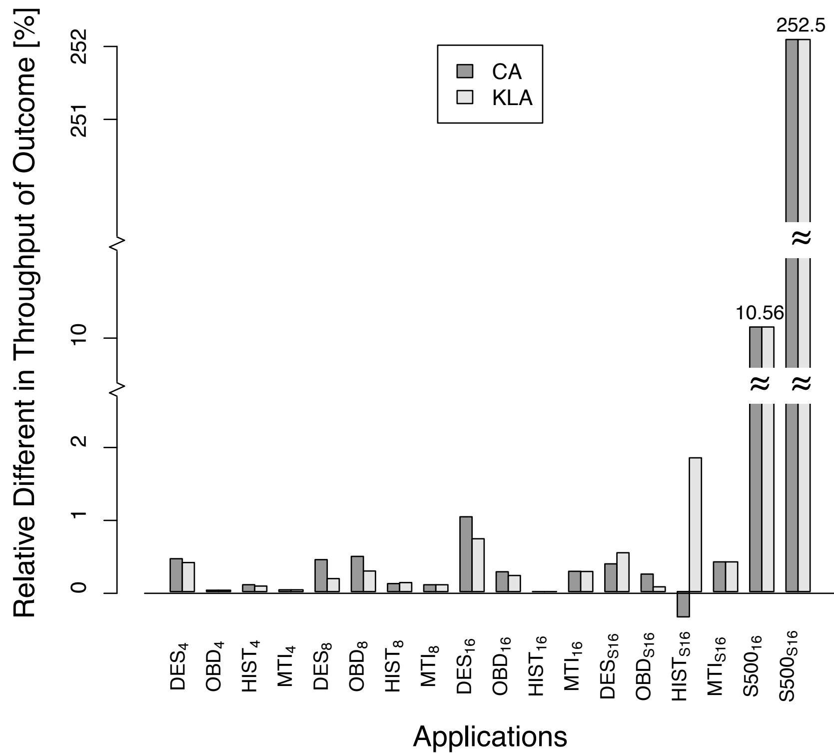
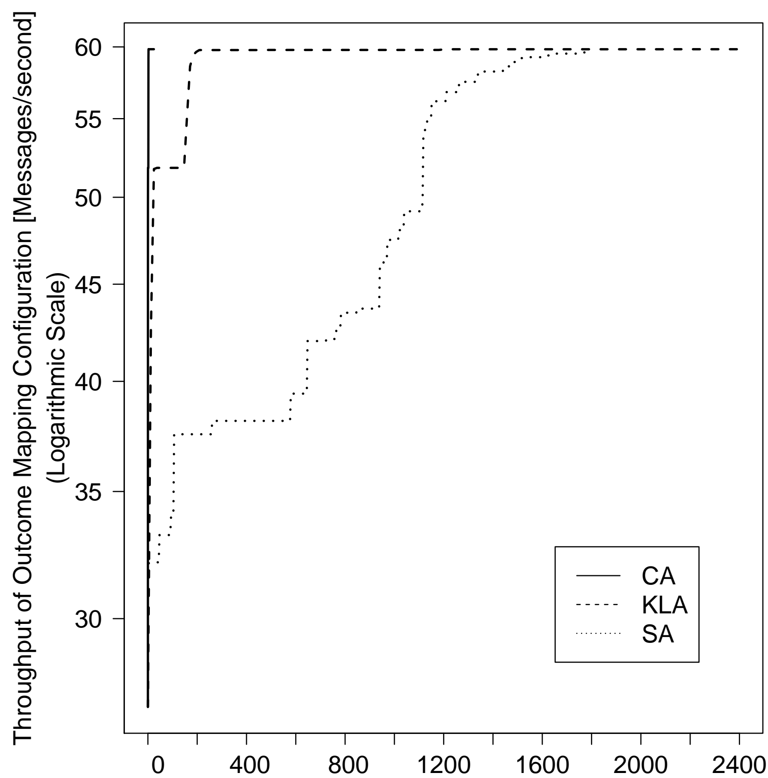
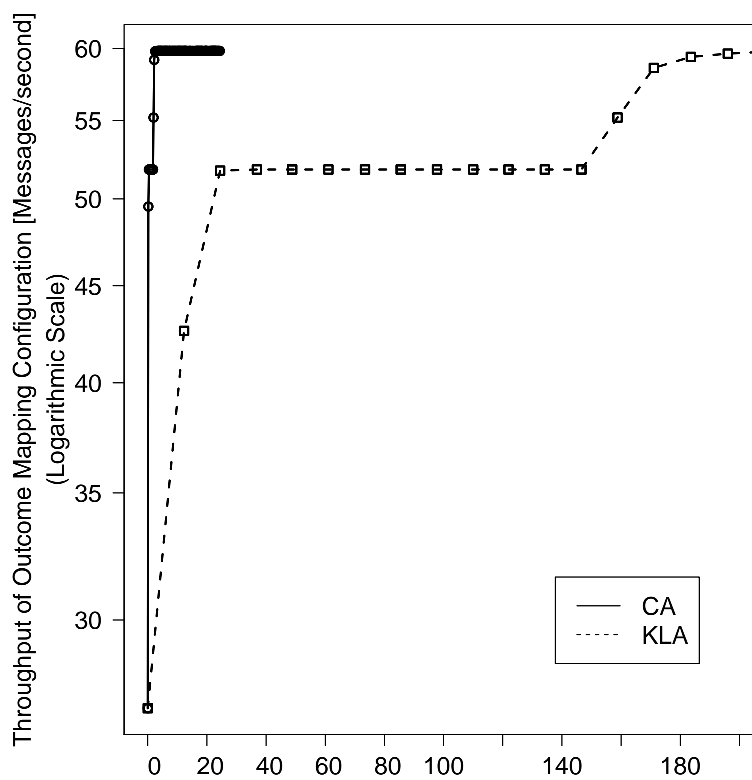


FIGURE 8.8: Achieved quality throughput estimates of 50-run CA and KLA compared to SA



(A) Convergence speed of SA, CA and KLA



(B) Convergence speed of CA and KLA (zoom of Figure 8.9a)

FIGURE 8.9: Convergence speed of SA, CA and KLA on MTI_{16} over time

Note that the temperature length is the number of iterations at each temperature. $neighbour_size$ is the number of neighbours of each state and it is also the number of new mapping configurations that can be generated by relocating one task to a new partition: $neighbour_size = |\mathcal{T}| \times (|\mathcal{R}| - 1)$.

Figure 8.7 and Figure 8.8 both show the comparison in terms of execution time and quality of the outcome mapping configuration. For benchmarks with a small number of edges and vertices, 50 runs of KLA is always faster than SA. For example, to partition DES_4 the 50 runs of KLA take around 0.06% of SA's execution time. The ratio is 1.4%, 0.2% and 0.11% for HIS_4, OBD_4 and MTI_4 respectively. When partitioning benchmarks with a large number of edges and vertices, 50 runs of KLA can take more time than SA. For example, it takes 242%, 567% and 683% of the execution time of SA to partition DES_{16}, $S500_{16}$ and $S500_{S16}$.

In contrast, the 50 runs of CA are always significantly faster than SA. It takes less than 5% of the execution time of SA for all benchmarks except for $S500_{16}$ and $S500_{S16}$. To partition these instances of S500, the 50 runs of CA take 9.3% and 15.7% of the execution time of SA for $S500_{16}$ and $S500_{S16}$, respectively.

For the quality of outcome mapping configurations, all three partitioning algorithms provide similar results for the non-synthetic benchmarks. The difference in throughput of these mapping configurations is less than 1%, except for DES_{16} where CA's result is 1.19% better than SA. For synthetic benchmarks $S500_{16}$ and $S500_{S16}$, CA and KLA provide mapping configurations with throughputs 10.56% and 252.6% higher than SA's outcome. Although one can search for a new set of parameters so that SA can provide a comparable result to CA and KLA, this may reduce the quality of the outcome mapping configuration or cause longer execution times for other benchmarks.

We also examine the convergence speed of all three partition algorithms. For all cases, CA is the fastest: it takes only a few milliseconds for small benchmarks like DES_4; and a few seconds for large ones like MTI_{16} and $S500_{16}$ to converge. Despite being a lot slower than CA, KLA still outperforms SA. For large benchmarks, KLA converges in a few hundred seconds while SA takes thousands of seconds. Figure 8.9a shows the convergence of these three algorithms on MTI_{16}. To better see the fast converge of CA, Figure 8.9b shows a zoomed version of Figure 8.9a.

8.5 Chapter Summary

Traditional graph-partitioning problems with the optimisation criterion being formed by the total number of cuts, are not applicable to the throughput optimisation of RSPs.

This chapter proposed two novel heuristic graph partitioning methods to partition the workload of RSPs to optimise throughput on heterogeneous distributed platforms. The first graph-partitioning algorithm is KLA, an adaptation of Kernighan-Lin. The second algorithm, called CA, narrows the search space compared to KLA, in particular by focusing on search points around the congestion, i.e. where the throughput is dimmed. Since KLA and CA are both local search heuristics, they have to be re-run multiple times in order to overcome local optima. We experimentally evaluated KLA and CA with five applications on four different platform configurations. We compared both methods with the generic meta-heuristic simulated annealing (SA) as a reference method. Even without restricting the available time for optimisation, both KLA and CA achieve at least as good throughput results as SA, sometimes even better. But the most important difference is their convergence speed. For small benchmarks KLA with its multiple reruns is up to 67 times faster than SA, but up to 7 times slower than SA for larger benchmarks. CA with its multiple reruns on the other hand is always orders of magnitudes faster than both KLA and SA, even for large graphs. Depending on the benchmark and platform, CA has been up to 300 times faster than KLA and up to 6000 times faster than SA. The outstanding speed of CA makes CA also potentially attractive for the re-partitioning of systems at runtime.

Chapter 9

An Efficient Execution Model for Reactive Stream Programs

This chapter combines all the work in previous chapters to create a new and efficient execution model for RSPs on two types of parallel platforms: i) uniform shared memory multi-core platforms, and ii) heterogeneous distributed systems consisting of uniform shared memory multi-core machines. The new execution model is facilitated with a monitoring framework (Chapter 6), a central-based scheduler for symmetric shared memory platforms (Chapter 7), and a partitioner to divide an RSP on distributed systems (Chapter 8).

9.1 Execution Model

9.1.1 Overview

This section gives an overview of the new execution model for RSPs. The execution model is designed to support RSPs on shared memory platforms and heterogeneous distributed platforms.

The design of the execution model is shown in Figure 9.1. This execution model includes three main layers: compiler, runtime system, and execution layer. Apart from these, the execution model is facilitated with two extra components: a monitoring framework as described in Chapter 6; and a CA partitioner as described in Chapter 8. In addition, the scheduler CS-dbp described in Chapter 7 is employed in the execution layer for local scheduling on each shared memory PE.

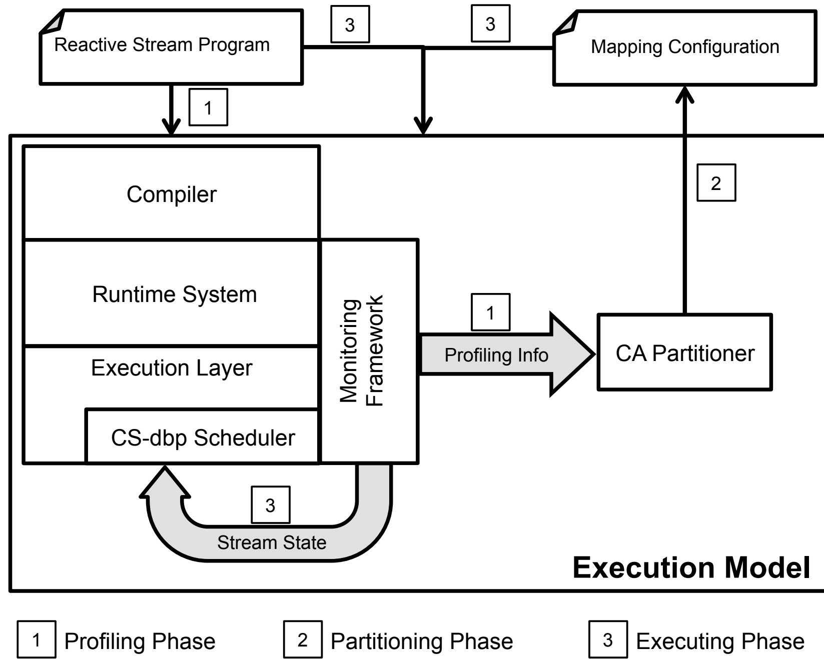


FIGURE 9.1: Overview of the new execution model

To execute an RSP on distributed platforms, the execution model requires the following three phases. A shared memory platform can be considered as a special case of distributed platforms of one PE. Deploying an RSP on such platforms does not require the first two phases.

- Profiling phase: The RSP is executed with sample data. The monitoring framework is set up to extract the RSP properties including the task weight and stream weight. The task-stream relations are also captured to create the task graph.
- Partitioning phase: The graph of the RSP is formed from the monitoring information in the profiling phase. This graph together with the graph of target platform are passed as inputs of the CA partitioner to generate a good mapping configuration for the RSP.
- Executing phase: The RSP is deployed with the mapping configuration generated in the previous phase to process real data. In this phase, the monitoring framework is set up to observe the stream state and delegate to the CS-dbp scheduler for the task priority calculation.

9.1.2 Integration into S-Net and LPEL

In this section, we present how the new execution model is integrated into the S-Net RTS and the LPEL execution layer. An overview of the integration is shown in Figure 9.2. Components of the monitoring framework are integrated into the S-Net RTS and LPEL. The monitoring framework writes monitoring information into text files. As these monitoring files contain raw data, a component called the Graph Constructor is employed to generate the graph of tasks.

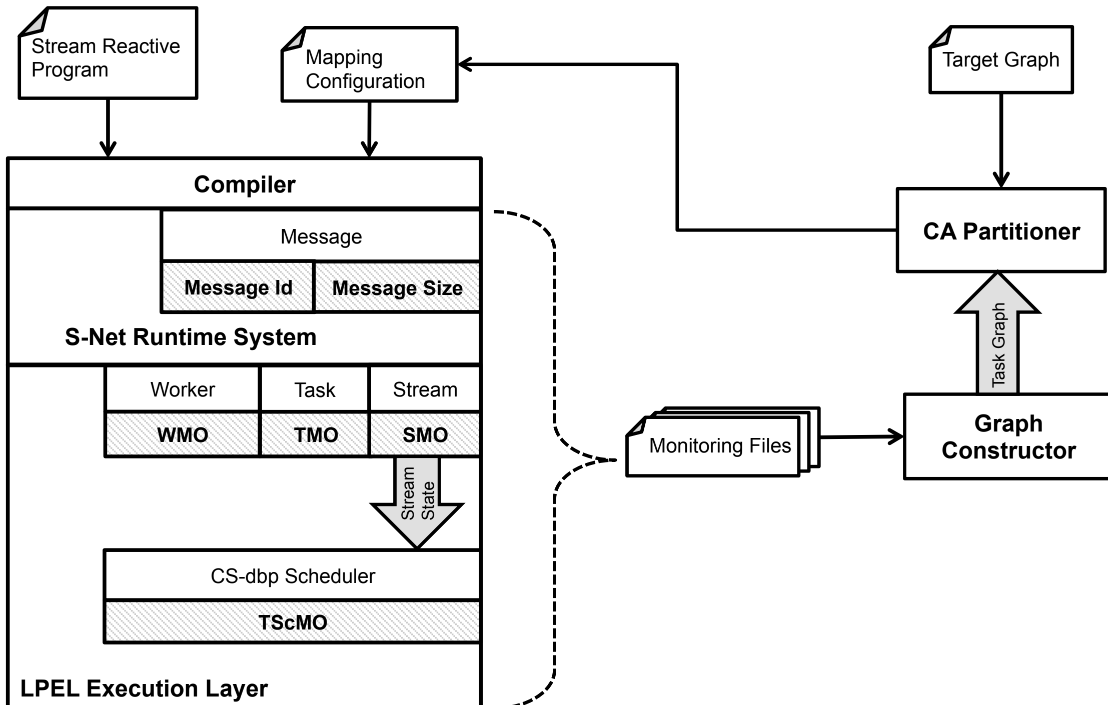
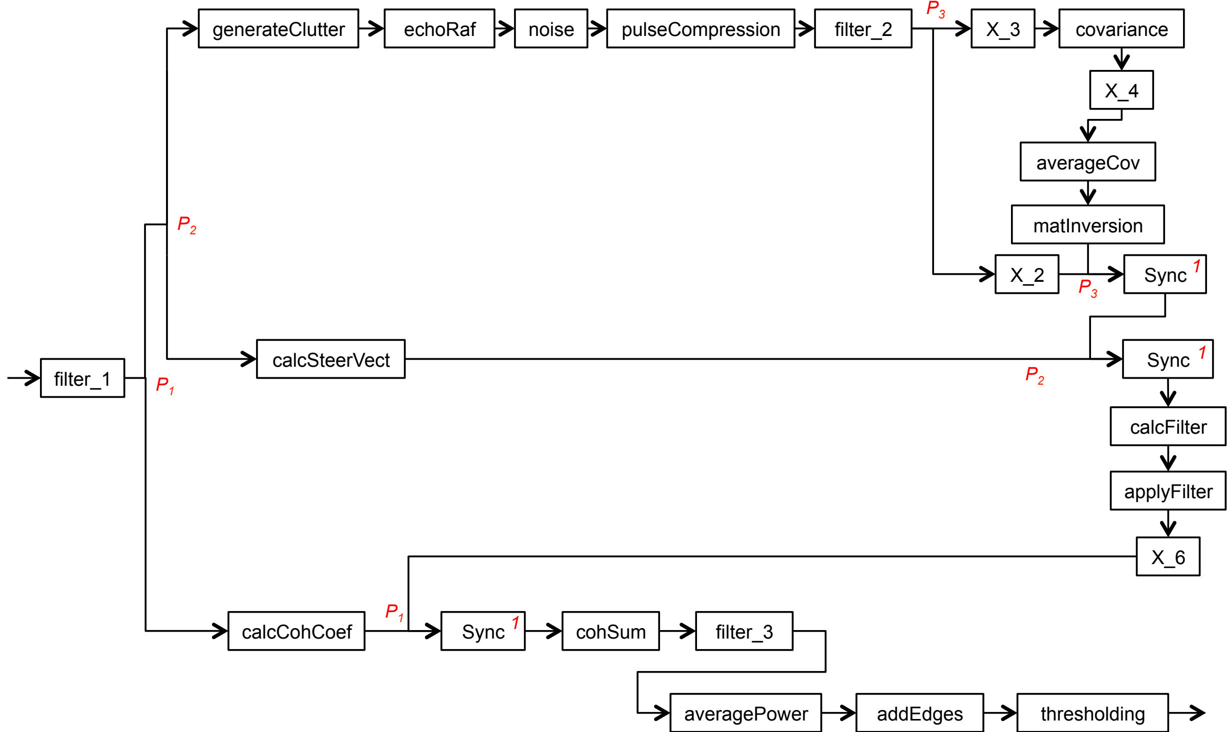


FIGURE 9.2: New execution model of S-Net using LPEL

The first job of the Graph Constructor is to create the structure of the task graph where vertices are tasks of the RSP and edges are streams connecting them. To do so, the Graph Constructor can extract the state of each stream to identify its task reader and task writer. This information shows the task-stream relations and therefore helps to construct the task graph. The second job of the Graph Constructor is to calculate the computational weight of each task, and the data weight of each stream. These calculations are performed based on the other monitoring information including MDG, message size, message events, and time scheduling events. The details of these calculations are presented in Section 6.2.2.

The task graph together with the target graph are passed to the CA Partitioner to generate the mapping configuration. The scheduler in LPEL is now replaced by the CS-dbp scheduler. The CS-dbp scheduler requires stream state information from the monitoring framework to operate.



[1] *"Sync" is a combination of a synchro-cell inside a serial replicator to merge the outputs from two branches of the prior parallel compositor*

P_1, P_2, P_3 *Parallel Compositors*

FIGURE 9.3: Main S-Net structure of the MTI application

The detailed implementations of the monitoring framework, CS-dbp scheduler and CA partitioner are presented in Section 6.3, Section 7.3, and Section 8.3.1 respectively. The RSP Graph Constructor has not been implemented and the task graph is currently generated manually.

9.2 Evaluation

9.2.1 Experimental Set Up

To evaluate the performance of the new execution model, we choose the Moving Target Indication (MTI) application as the use case. To increase the parallelism, the application is implemented with a parallel replication whose operand is a main S-Net structure that performs the MTI application's function. This structure is shown in Figure 9.3 and the detailed implementation can be found in [PHG+10]. A number of instances of this structure are generated dynamically depending on the number of machines available.

The experiment platform is a cluster of 16 machines, each of which has 24Gb memory and 2 Xeons E5520 CPUs. Each CPU has 4 cores. The machines are connected via a 4xDDR Infiniband network where the traffic between each pair of machines is guaranteed for a full bandwidth of 16 Gbits/s.

Since all the machines on the platform have the same configuration, we first run the MTI application with sample data on one machine in the platform. We then use the monitoring framework to obtain the task graph. With the monitoring framework, we can also extract the task weight and stream weight. These are the computational cost of each task and the communication cost between each pair of tasks respectively.

9.2.2 Performance on Distributed Platforms

In this section, we evaluate the performance of the MTI application on the target of 1, 2, 4, 8 and 16 machines. For each target, one of its machine needs to dedicate 2 cores to simulate the source and sink for the MTI application.

The configuration of each target together with the task graph of the MTI application are used as the input for the CA partitioner. The output is the mapping configuration for each target.

Figure 9.4 shows the actual throughput of the MTI application compared to the estimated throughput from the CA partitioner. Figure 9.4 shows that the throughput of MTI scales up when the number of machine increases. However, the throughput is matches the estimation only in the case of one machine. For other cases, it is lower compared to the estimated value, and the difference seems to increase when the number of machines increases.

One reason for this phenomena is that the LPEL conductor and workers do not have exclusive usage of the CPU cores when deploying S-Net on a distributed platform. As presented in Section 2.5.4, the distributed S-Net RTS creates three extra threads for the input manager (IM), output manager (OM) and data fetcher (DF). These threads are run independent from LPEL conductor and workers. They also share the same set of CPU cores with the LPEL conductor and the workers. This invalidates the information of task weight and therefore varies the actual throughput.

To avoid this phenomena, each machine of the distributed platform dedicates three CPU cores for the IM, OM and DF. This guarantees that each LPEL conductor and worker has its own exclusive CPU core. The throughput of the MTI application on this new platform is shown in Figure 9.5. The different between the actual throughput value and

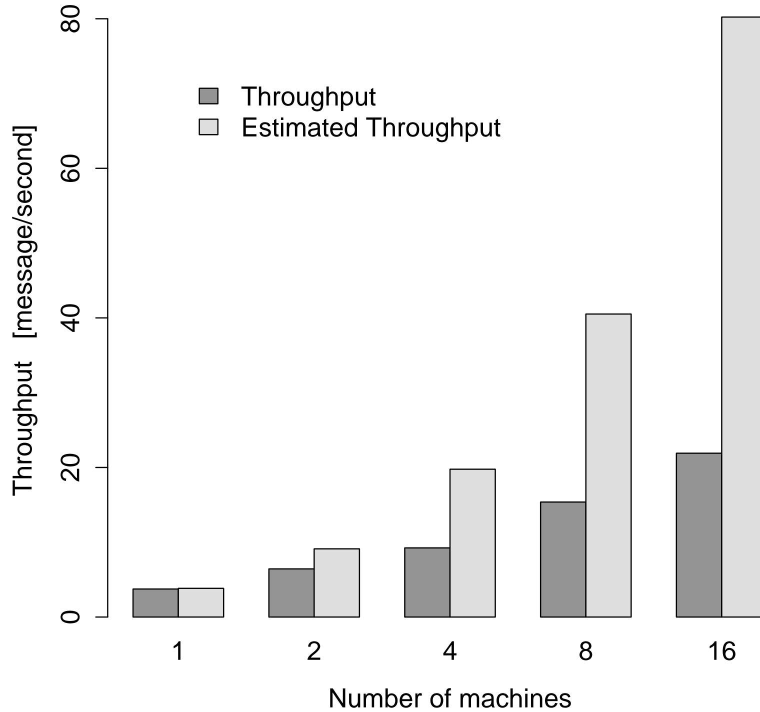


FIGURE 9.4: Throughput of the MTI application: actual value vs estimation of the CA partitioner

the estimated value has been improved although it is still high in the cases of 8 and 16 machines. There are two reasons behind this.

Firstly, the computational weight of each task when deploying on distributed platforms may increase in an unpredictable way compared to when deploying on shared memory platforms. As described in Section 2.5.4, when running on a distributed platforms, the distributed S-Net RTS does not send actual data via streams across the border of PEs. Instead only the reference of the data is included in the message and when an RC needs the data, it will send a fetch request. The fetch request is sent inside the RC by using the function *MPI_Send* which operates in *blocking mode*. This means the function call *MPI_Send* does not return until the fetch request is received by the destination PE. During this time, the task associated with the RC performs no computation but does not release the worker so that another task can be executed. This time period is therefore counted as the execution time of the task. Note that on each machine, only IM listens and responds to the MPI communications. The length of this idling period depends on the status of the MPI buffer on the destination PE. This value therefore varies for different fetch requests and is unpredictable. Figure 9.6 shows the difference in task weight of boxes in the MTI application when running on a single machine target where

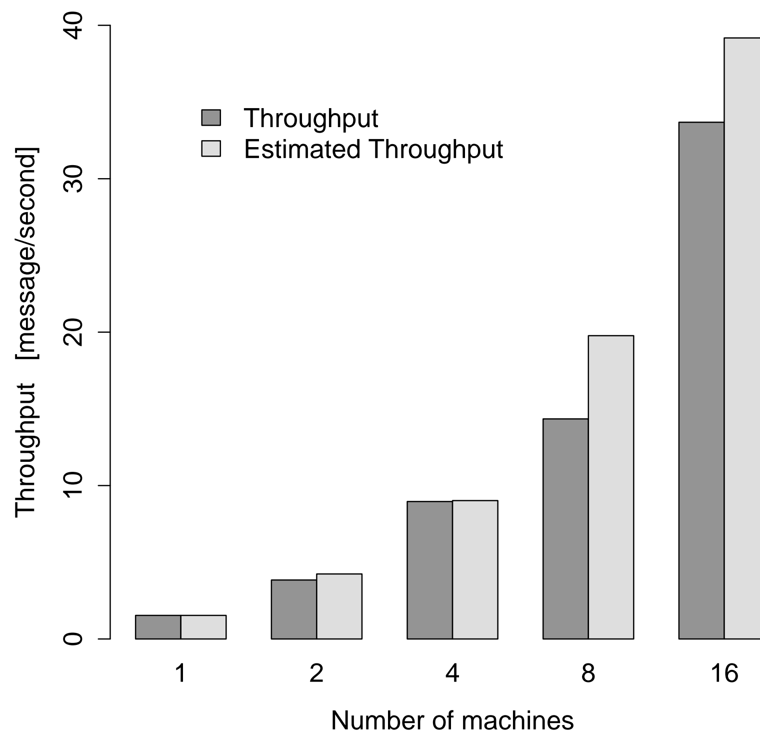


FIGURE 9.5: Throughput of the MTI application when LPEL conductor and workers run on exclusive CPU cores

no MPI function is invoked, and when running on a 16-machine target. The difference in task weight is significant for some boxes such as 78.4 seconds for *applyFilter*, 42.9 seconds for *echoRaf*, and 33.2 seconds for *calcFilter*.

Since the distributed S-Net RTS changes the behaviours of tasks, their computation weight extracted by the monitoring framework is no longer accurate. With a larger number of machines, it is likely to have more fetch requests and that decreases the accuracy. The difference between the actual and estimated throughput values is therefore more significant when the number of machines increases.

Secondly, the relative idling time of the CS-dbp scheduler varies between deployments on shared memory and distributed platforms. According to Section 8.2, the CA partitioner designed based on an assumption that the local scheduler of each PE has predictable relative idling time $\widetilde{W}$ and relative overhead time $\widetilde{O}$. In the case where the CS-dpb scheduler is used, the relative overhead time is fixed. It is equal to 1 as one CPU core is dedicated for the conductor. The relative idling time on the other hand is not guaranteed to be fixed. CS-dbp is a heuristic approach that aims to minimise the relative idling time. In this scheduler, a worker is idle when it has to wait for a new task from the conductor. With a sufficient input arrival rate, we can consider that there are always

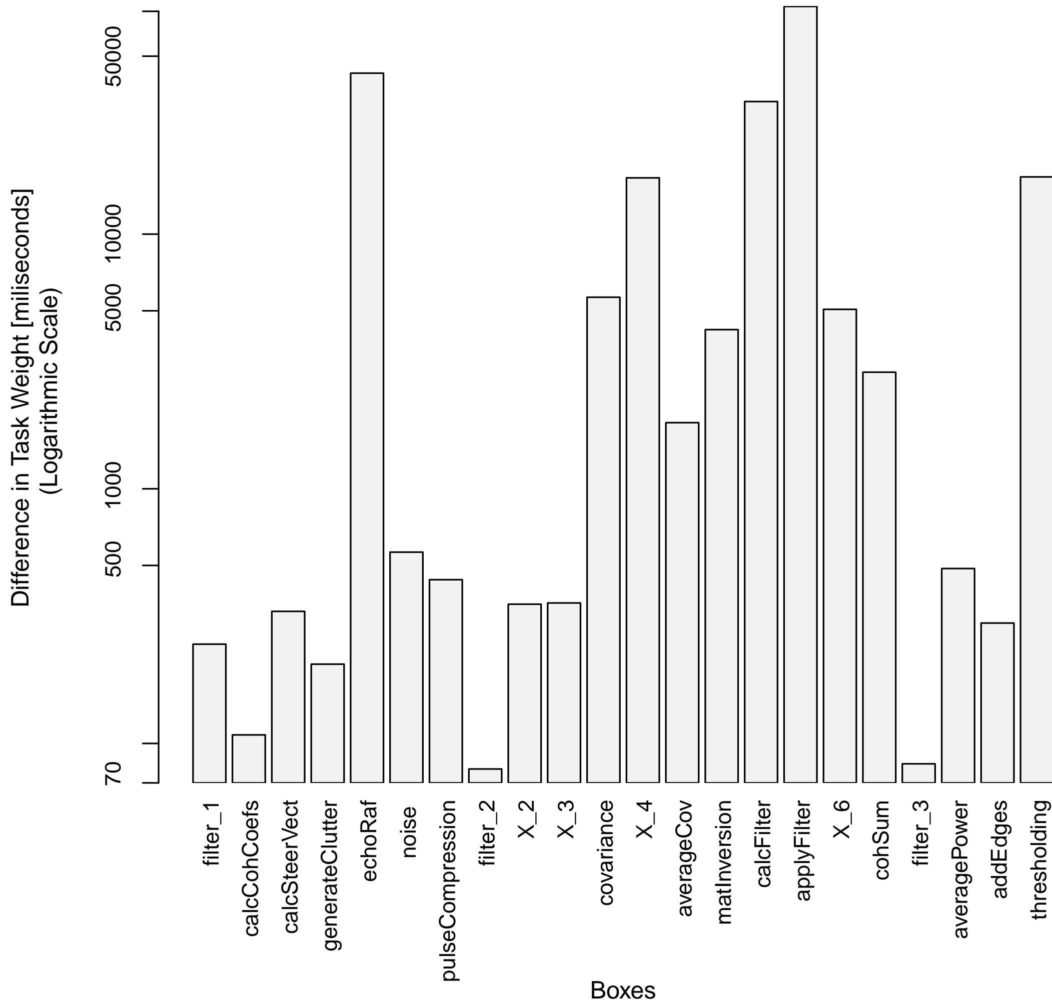


FIGURE 9.6: Difference in task weight of MTI on 1-machine and 16-machine targets

ready tasks in the CTQ. A worker therefore would become idle only when the conductor is busy with task requests from other workers, or busy updating the task priority in the CTQ. The idling time therefore depends on the number of tasks and the number of workers. With a large number of workers, it is more likely that two or more workers happen to request tasks at the same time. This appears not to be an issue here as the number of workers is 2 for the first machine where source and sink are simulated; and is 4 for other machines. Since the CTQ is implemented by a heap structure, updating a task priority requires a complexity of $O(\log(|RT|))$ where $|RT|$ is the number of ready tasks. With a larger number of ready tasks the update of the task priority will take more time, although it is logarithmic. In this case, it will take a longer time to update the task priority. When running on a platform with a large number of machines, more instances of the primary S-Net structure in MTI are generated. This creates more tasks and this is likely to increases the relative idling time.

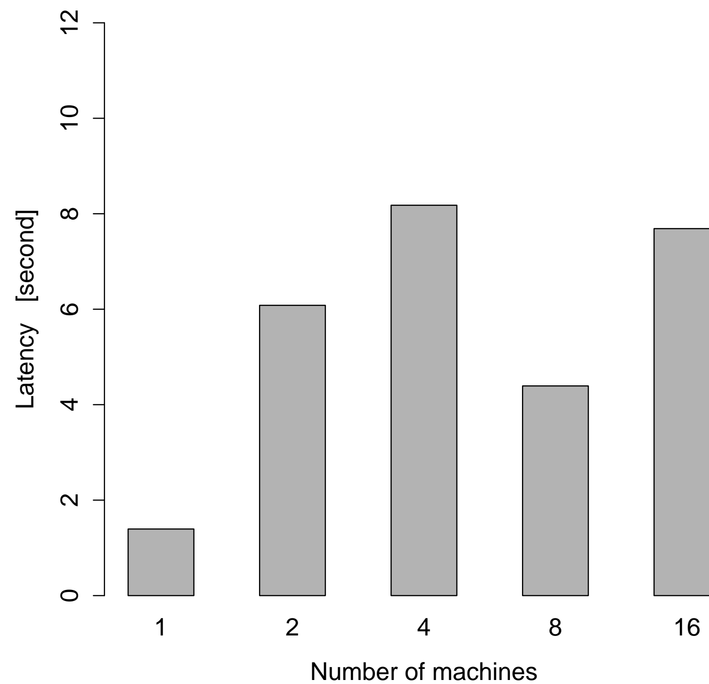


FIGURE 9.7: Average latency of MTI on distributed platforms when LPEL conductors and workers run on exclusive CPU cores

The CA partitioner has been designed for throughput optimisation without considering the latency. However, we include here the latency result on distributed platforms for reference. Optimising latency in the partitioning algorithms will be discussed in the future work. Figure 9.7 shows the latency of MTI on distributed platforms of 1, 2, 4, 8, and 16 machines. In this experiment, each machine dedicates 3 CPU cores for IM, OM, and DF. In addition, the first machine also dedicates 2 more CPU cores for source and sink simulation.

9.2.3 Performance on shared memory platforms

In this section, we perform the experiment to evaluate these metrics of MTI on shared memory platforms. The deployed platform is one machine from the distributed platform described in the previous section. The machine has 8 CPU cores and 2 of them are used to simulate the source and sink. Figure 9.8 shows the performance of MTI on the shared memory platform of 2, 3, 4, 5, and 6 CPU cores. Both the throughput and latency are improved when the number of CPU cores increases from 2 to 5. While throughput scales almost in a linear way, the latency does not scale that well because of the limited concurrency available in the MTI benchmark. As shown in Figure 9.3, there are only three parallel compositions P_1, P_2, and P_3, where computation of an

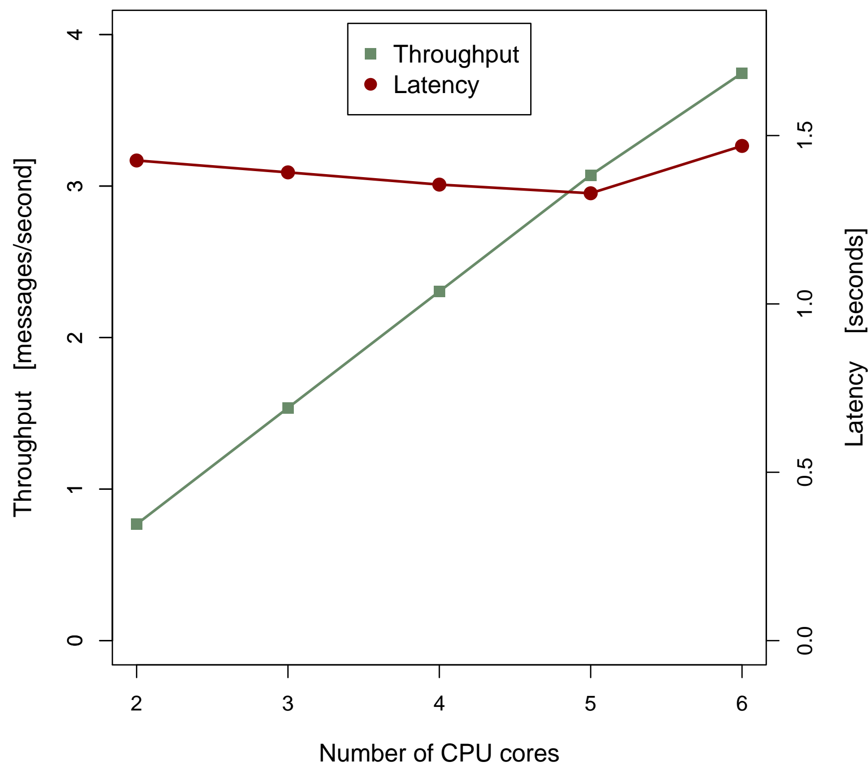


FIGURE 9.8: Performance of MTI on shared memory platform

external input message can be performed concurrently. All these parallel compositors connect with a *Sync* structure which merges the output of their branches together. This requires the output of their branches to wait for each other before being moved further. The maximum concurrency in processing one external input message occurs when two branches of each parallel compositors are executed at the same time. As shown in Figure 9.3, one of two branches in all P_1, P_2, and P_3 contains only one box. This in fact limits the level of concurrency. Therefore, increasing the number of cores can improve the latency only by a small factor.

With 6 CPU cores, the throughput is still increased but not at the same scale. The latency in this case is worse, it is 10% higher than the latency in case of 5 CPU cores. This is because of the bottleneck at the conductor. As the number of CPU cores increases, there are more workers while there is only one conductor. With more workers, there is a higher chance that multiple workers send their requests to the conductor at the same time. Only one request is served while others have to wait. This increases the idling time of workers. Thus, it degrades the latency and slightly reduces the scaling of throughput.

9.3 Chapter Summary

By combining the work from the previous chapters, this chapter described an efficient execution model for RSPs at both conceptual and implementation levels. The execution model makes use of the CA algorithm in Chapter 8 to map the RSP onto distributed platforms and the centralised scheduler CS-dbp (described in Chapter 7) for scheduling the RSP on each shared memory PE. The execution model also makes use of the monitoring framework described in Chapter 6 to extract RSP properties as the input of the CA algorithm; and to provide the stream state for calculating task priority in CS-dbp.

In addition, this chapter used an industrial application to evaluate the new execution model. The experiment results shows unexpected throughput compared to the estimated value by the CA algorithm. This is explained by the implementation of the distributed S-Net RTS, which alters the task behaviours; and the bottleneck of the conductor in CS-dbp. This bottleneck also limits the scalability of the CS-dbp scheduler.

Chapter 10

Conclusion and Outlook

This chapter concludes the thesis by summarising the main features and contributions. It also gives an overview of future research directions in scheduling RSPs on parallel platforms.

10.1 Thesis Summary

This dissertation presented novel approaches for efficiently scheduling RSPs for both uniform shared memory multi-core platforms, and heterogeneous distributed platforms consisting of multi-core machines with uniform shared memory. The challenge is that implicit synchronisation via stream communication together with variable behaviour of computational nodes make it intricate to analyse and regulate the performance of RSPs.

As discussed earlier in Chapter 3, most of the existing work in the stream programming literature focuses on SDF programs, a simplified stream programming model with static properties. This work, on the other hand, focuses on the general class of RSPs. As presented in Chapter 2, these RSPs can have functional or non-functional node computation, constant or variable node behaviour, synchronous or asynchronous inter-node communications, and static or dynamic program structures. Existing work is therefore not suitable for this type or RSPs. By analysing the performance of general RSPs, we identified factors that affect the performance. We then introduced methods to maximise the performance and integrate it into a new execution model for RSPs.

The dissertation aims to answer the research question proposed in Chapter 1:

What is an efficient execution model for general reactive stream programs?

This question is fractured into the following sub-questions:

1. Which behavioural factors have influences on the performance of RSPs?
2. How can these behavioural factors be captured?
3. What are the strategies to optimise the performance of RSPs on parallel platforms?

To answer the above sub-questions, the dissertation has made following contributions:

Throughput and latency in RSPs: To answer the first sub-question, the contribution in Chapter 5 first provided a study of the performance in terms of throughput and latency of RSPs. The study recognised border limits of both throughput and latency. The chapter also presented quantitative formulae of throughput and latency within the underuse and operational ranges of the input arrival rate. The quantitative formulae help to identify the behaviour factors that have influences on the performance of RSPs.

Capturing the behaviour of RSPs: Next, to answer the second sub-question, we have presented in Chapter 6 a monitoring framework to capture essential behavioural information of the RSP. The chapter included both the concept design and implementation of the monitoring frame work. The monitoring information was shown to be useful in different scenarios including performance calculation, property extraction, automatic load balancing and bottle neck detection. The experimental results in Chapter 6 show that the overhead of this monitoring framework is negligible for application implementations that are suitable for the stream programming model.

Using properties of RSPs to derive task priority on shared memory platforms: Addressing the third sub-question of optimising the performance of RSPs on uniform shared memory platform, Chapter 7 introduced novel approaches of utilising the RSP properties to derive scheduling priorities. Based on the performance analysis in Chapter 5, we determined two characteristics of RSPs to form the task priority for optimising the performance in terms of throughput and latency. These characteristics are the data demand on stream communications, and the structural position of tasks within the RSP. These characteristics are used to construct two schedulers CS-dbp and CS-pbp for RSPs on uniform shared memory platforms. While the CS-dbp makes use of the notion of data demands on stream communication to derive the task priority, the CS-pdp takes advantage of the structural position of each task in the stream graph. Although they achieve similar performance, CS-dbp is more beneficial as its implementation is independent from the RTS, and its applicability is not restricted to RSPs with single entry and/or single

exit nodes. The experimental results in Chapter 7 show that CS-dbp surpasses the default scheduler of LPEL in terms of both throughput and latency. CS-dbp and CS-pdp utilise the data demands and the structural positions to define the task priority. Since these features are observable during runtime, these schedulers are applicable for general RSPs, especially with variable node behaviour and dynamic program structures.

Exploiting graph partitioning to map RSPs onto heterogeneous distributed systems: Also addressing the third sub-question, Chapter 8 aimed to optimise the throughput of RSPs on distributed systems. Based on the performance analysis in Chapter 5, we presented in Chapter 8 two new heuristic partitioning algorithms to efficiently map RSPs onto heterogeneous distributed platforms. These are Kernighan-Lin Adapted (KLA), and Congestion Avoidance (CA). In contrast to traditional partitioning algorithms where the total cut is the main optimisation target, these two aim to optimise the throughput of RSPs where individual cuts play an important role. KLA is designed as an adaptation from the famous Kernighan Lin algorithm. This works well although it performs very slowly. CA is an improvement of KLA by narrowing the search space to points around the congestion points. This helps to speed up CA and makes it more attractive for the consideration of online repartition. Chapter 8 compares CA with Simulated Annealing (SA), which is a generic multi-parameter optimisation algorithm. CA is significantly faster than SA while producing mapping configurations with the same quality. Depending on the benchmark, CA can be 4 to 300 times faster than KLA, and 1.6 to 6000 times faster than SA. Since CA and KLA are off-line approaches, they are only applicable to RSPs with relatively stable node behaviour and relatively static program structures. In cases where these properties are highly variable, CA and KLA can still be used to generate the initial mapping configuration. During runtime, an adaptation mechanism is required for repartitioning the RSP when necessary.

Integration into a new efficient execution model for RSPs: Finally, we presented a complete integration of the monitoring framework, the CS-dbp scheduler, and the CA partitioner to form an integrated execution model. We also showed the three phases to deploy an RSP using the new execution model. These include a profiling phase to extract properties of the RSP and form the input for the CA partitioner; a partitioning phase to generate the mapping configuration for the RSP on the distributed system; and an executing phase to execute the RSP with the generated mapping configuration. The experiment results in Chapter 9 have shown good achievements of the new execution model both on shared memory platforms and distribution systems. The composition of the presented monitoring strategy, scheduling techniques and mapping algorithms towards a new execution

model finally provides an answer to the overall research question: 'What is an efficient execution model for general RSPs?'.

10.2 Outlook

Within this thesis, we have presented the current state of our research in optimising the performance of RSPs on parallel platforms. We have introduced approaches to exploit properties of RSPs to effectively schedule RSPs on shared memory platforms; as well as strategies to map RSPs onto distributed platforms. Although the results presented in previous chapters have demonstrated the effectiveness of these approaches, it could be further developed in a number of ways:

- **Further study of RSP properties in deriving scheduling priorities**. Chapter 7 has demonstrated the usages of two different RSP properties to derive the task priority in the centralised scheduling approach. These properties include data demand of stream communication; and the structural position of tasks within the RSP. While the first is dynamic and requires perpetual evaluation, the second is fixed and can be generated on task creation. However, the task priority based on the structural position is a vector describing the path from the entry task to the task itself. Comparing the priority between tasks in this case is therefore slow. To overcome this problem, one could investigate a new way to represent the structural position so that it speeds up the priority comparison. Also, there may be other ways of defining the task priority which do not require perpetual evaluation while allowing fast priority comparisons.
- **Conquering the bottleneck in centralised scheduling**. The experimental results in Chapter 7 and Chapter 9 have shown that throughput and latency do not scale well with high numbers of workers. This occurs with most of the centralised scheduling approaches. The reason behind this is the bottleneck of the conductor. When a conductor receives a task request from a worker, it needs to pick a new task, and send it to the worker. The conductor is also responsible for updating the central task queue. With the current implementation, a worker requests a new task when it finishes its work for the current task. This causes a delay when a worker requests a task while the conductor is busy either serving task requests from other workers or updating the CTQ. To overcome this problem, there are some directions to be considered:
  - *Fast updating of the central task queue.* As described in Chapter 7, the central task queue is implemented by a heap data structure to store only ready tasks.

To update the task queue, it requires the complexity of $O(\log(|RT|))$, where $|RT|$ is the number of ready tasks. This implementation fits well when a task's priority is dynamically changed during its life time. In the case where the task priority is fixed, there are potential data structures that can help to shorten the time to update the CTQ.

- *Efficient conductor-worker communication protocol.* Instead of requesting a new task after finishing the work of the current task, a worker could request a new task earlier. This gives the conductor an appropriate amount of time to pick the task and update the CTQ. An alternative way is that the conductor can predict when a worker is close to finish its current work and send a new task beforehand. This would allow the conductor the flexibility to prepare for the new task for each worker. This technique has a risk of causing a worker to wait when the conductor can not predict accurately when the worker finishes its current work. Therefore, it requires methods for precisely predicting the finishing time of workers. The disadvantage of these two proposed techniques is that they relax the order of task executions. Since the new task for a worker is chosen before it is actually needed, other tasks with higher priority becoming available after that will be delayed for execution. This can be improved by appropriate strategies to allow the conductor to replace the new task of each worker. Alternatively, the conductor can send the new task with higher priority to a worker. The worker then can choose to execute the highest priority task and return other tasks back to the conductor.

- **Considering latency when partitioning RSPs**. Chapter 8 has presented partitioning algorithms to map RSPs onto distributed platforms for throughput optimisation. It has shown that the throughput is improved when deploying the RSP on larger numbers of PEs. It is more complicated when dealing with latency as the level of concurrency in processing one external input message depends on the implementation of the RSP. With a low level of concurrency, increasing the number of PEs does not help to improve the latency. It however is more likely to burden the latency as the cost of inter-PE communication is significant. The remaining question is how to minimise the effect on the latency. According to Equation 5.6, the latency is proportional to the number of external input messages currently being processed ($\overline{M_{cp}}$). Mapping nodes of an RSP onto different PEs breaks the notion of data demands on streams across PEs. The inter-PE communication channels provide extra space for these streams to store more partly processed external input messages and therefore increase $\overline{M_{cp}}$. As a result, the increment of $\overline{M_{cp}}$ is controlled by the number of cuts among partitions. It would be promising to integrate this optimising parameter into the proposed partitioning algorithms.

It would be also interesting to see if there is a conflict between this new parameter and the existing ones.

- **Online adaptation by repartitioning**. As we mentioned in Chapter 8, the properties of the RSP graph are statistically derived from the monitoring information. As these properties are not static, they can change during runtime and the mapping configuration as a consequence might be no longer efficient. In this case, online adaptation strategies are required. Since the CA algorithm is shown to be very fast, it could be a promising direction for repartitioning the RSP during runtime. Of course, it would need some modifications in the optimisation target to consider the cost of migrating tasks to different PEs.

Bibliography

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, PedroR. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, KimG. Larsen, M.Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In Franck Cassez, Claude Jard, Brigitte Rozoy, and MarkDermot Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42787-2. Available from: http://dx.doi.org/10.1007/3-540-45510-8_4, doi:10.1007/3-540-45510-8_4.

[ABF+10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. ISSN 1532-0634. Available from: http://dx.doi.org/10.1002/cpe.1553, doi:10.1002/cpe.1553.

[ADA79] W. B. Acherman, J. B. Dennis, and William B Ackerman. Val- oriented algorithmic language, preliminary reference manual. Technical report, Cambridge, MA, USA, 1979.

[AHL07] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 112–120. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-602-8. Available from: http://doi.acm.org/10.1145/1229428.1229448, doi:10.1145/1229428.1229448.

[AKY99] Charles J Alpert, Andrew B Kahng, and So-Zen Yao. Spectral partitioning with multiple eigenvectors. *Discrete Applied Mathematics*, 90(1–3):3 – 26, 1999.

[ARV09] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):5:1–5:37, April 2009. ISSN 0004-5411.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20, July 1977. ISSN 0001-0782.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.

[BBHL02] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, and Edward A. Lee. Readings in hardware/software co-design. chapter Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms, pages 452–464. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-55860-702-1. Available from: http://dl.acm.org/citation.cfm?id=567003.567042.

[Ber89] Gérard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP Congress*, pages 11–17, 1989.

[BF81] R. M. Bryant and Raphael A. Finkel. A stable distributed scheduling algorithm. In *Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981*, pages 314–323, 1981.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786. ACM, New York, NY, USA, 2004. doi:10.1145/1186562.1015800.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995. ISSN 0362-1340. doi:10.1145/209937.209958.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. ISSN 0004-5411. doi:10.1145/324133.324234.

[BS73] Fischer Black and Myron Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973. ISSN 00223808. Available from: http://dx.doi.org/10.2307/1831029, doi:10.2307/1831029.

[BS81] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194. ACM, New York, NY, USA, 1981. ISBN 0-89791-060-5. doi:10.1145/800223.806778.

[BT93] Dimitris Bertsimas and John Tsitsiklis. Simulated Annealing. *Statistical Science*, 8(1):10–15, 1993.

[Bur75] William H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25, 1975.

[cA99] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[CC09] Rebecca L. Collins and Luca P. Carloni. Flexible Filters: Load balancing through backpressure for stream programs. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 205–214. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-627-4. Available from: http://doi.acm.org/10.1145/1629335.1629363, doi:10.1145/1629335.1629363.

[Cha98] Bradford L. Chamberlain. Graph partitioning algorithms for distributing workloads of parallel computations. Technical report, 1998.

[CRA09] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 57–66. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-626-7.

[Dav78] A. L Davis. The architecture and system method for DDM1: A recursively structured data-driven machine. In *5th Annual Symp. on Computer Architecture*, 1978.

[Den77] JackB. Dennis. A language design for structured concurrency. In JohnH. Williams and DavidA. Fisher, editors, *Design and Implementation of Programming Languages*, volume 54 of *Lecture Notes in Computer Science*, pages 231–242. Springer Berlin Heidelberg, 1977.

[Den80] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980. doi:10.1109/MC.1980.1653418.

[Den95] Jack B. Dennis. *Stream data types for signal processing*, pages 87–102. IEEE Computer Society Press, 1995.

[DES77] DES. Data Encryption Standard. In *FIPS PUB 46-3, Federal Information Processing Standards Publication*, 1977.

[DO87] Fred Douglis and John K. Ousterhout. Process migration in the sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, September 1987*, pages 18–27, 1987.

[dOCLB10] Pablo de Oliveira Castro, Stéphane Louise, and Denis Barthou. Automatic mapping of stream programs on multicore architectures. In *International Workshop on Compilers for Parallel Computers*, 2010.

[DPSW98] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Aspect radio for mesh partitioning. In *Euro-Par*, pages 347–351, 1998.

[DPSW00] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive fem. *Parallel Comput.*, 26(12):1555–1581, November 2000.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Book, 2004.

[ELLSV99] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *PROCEEDINGS OF THE IEEE*, pages 366–390, 1999.

[ELZ86] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12 (5):662–675, May 1986. ISSN 0098-5589. Available from: http://dl.acm.org/citation.cfm?id=5527.5535.

[ET63] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *Electronic Computers, IEEE Transactions on*, EC-12(6):755–773, 1963.

[FFJ90] John Franco, Daniel P. Friedman, and Steven D. Johnson. Multi-way streams in scheme. *Comput. Lang.*, 15(2):109–125, April 1990. ISSN 0096-0551. Available from: http://dx.doi.org/10.1016/0096-0551(90)90014-G, doi:10.1016/0096-0551(90)90014-G.

[FKBS11] Sardar M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. Orchestration by approximation: Mapping stream programs onto multicore

architectures. *SIGPLAN Not.*, 46(3):357–368, March 2011. ISSN 0362-1340. doi:10.1145/1961296.1950406.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181. IEEE Press, Piscataway, NJ, USA, 1982. ISBN 0-89791-020-6.

[FRS+97] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 1–34. Springer-Verlag, London, UK, UK, 1997. ISBN 3-540-63574-2.

[GAF+09] Buğra Gedik, Henrique Andrade, Andy Frenkiel, Wim De Pauw, Michael Pfeifer, Paul Allen, Norman Cohen, and Kun-Lung Wu. Tools and strategies for debugging distributed stream processing applications. *Softw. Pract. Exper.*, 39:1347–1376, November 2009. ISSN 0038-0644. Available from: http://dx.doi.org/10.1002/spe.v39:16, doi:http://dx.doi.org/10.1002/spe.v39:16.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-102-6. Available from: http://doi.acm.org/10.1145/1376616.1376729, doi:10.1145/1376616.1376729.

[GG13] Bert Gijsbers and Clemens Grelck. An efficient scalable runtime system for macro data flow processing using S-Net. *International Journal of Parallel Programming*, pages 1–24, 2013. ISSN 0885-7458. doi:10.1007/s10766-013-0271-8.

[GJP12] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net: Cluster and grid computing without the hassle. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 410–418. IEEE Computer Society, Washington, DC, USA, 2012. ISBN 978-0-7695-4691-9. doi:10.1109/CCGrid.2012.140.

[GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976. ISSN 0304-3975.

[GO10] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurr. Comput. : Pract. Exper.*, 22(6):736–748, April 2010. ISSN 1532-0626. doi:10.1002/cpe.v22:6.

[Gor10] Michael I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2010. Available from: http://groups.csail.mit.edu/commit/papers/10/mgordon-phd-thesis.pdf.

[GP11] Clemens Grelck and Frank Penczek. Implementation architecture and multithreaded runtime system of S-NET. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 60–79. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-24451-3.

[GSS08] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162. New York, NY, USA, 2006. ISBN 1-59593-451-0. Available from: http://doi.acm.org/10.1145/1168857.1168877, doi:10.1145/1168857.1168877.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30(5):291–303, October 2002. ISSN 0163-5964. Available from: http://doi.acm.org/10.1145/635506.605428, doi:10.1145/635506.605428.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010. doi:10.1002/cpe.1556.

[Hal84] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional*

Programming, LFP '84, pages 9–17. ACM, New York, NY, USA, 1984. ISBN 0-89791-142-3. doi:10.1145/800055.802017.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems - a tutorial and commented bibliography. In *In Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), LNCS 1427*, pages 1–16. Springer Verlag, 1998.

[HCK+09] Amir Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric M. Rabbah, Trevor N. Mudge, and Scott A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, pages 214–223, 2009.

[HK06] L. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 11(9):1074–1085, November 2006. ISSN 0278-0070.

[HL95a] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. ACM, New York, NY, USA, 1995. ISBN 0-89791-816-9. doi:10.1145/224170.224228.

[HL95b] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. ACM, New York, NY, USA, 1995. ISBN 0-89791-816-9. Available from: http://doi.acm.org/10.1145/224170.224228, doi:10.1145/224170.224228.

[HMSM06] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. TAUg: Runtime global performance data access using mpi. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 313–321. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3-540-39110-X, 978-3-540-39110-4. doi:10.1007/11846802_44.

[HP85] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15181-8.

[hSOH11] Tae ho Shin, Hyunok Oh, and Soonhoi Ha. Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In *Design Automation Conference (ASP-DAC), 2011 16th Asia*

and South Pacific, pages 165–170, Jan 2011. ISSN 2153-6961. doi:10.1109/ASPDAC.2011.5722178.

[ILO14] ILOG. CPLEX math programming engine, July 2014. Available from: http://www.ilog.com/products/cplex/.

[JAMS91] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Oper. Res.*, 39(3): 378–406, May 1991. ISSN 0030-364X. doi:10.1287/opre.39.3.378.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004. ISSN 0360-0300.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland, Amsterdam, Stockholm, Sweden, Aug 1974.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KHP+09] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. COLA: Optimizing stream processing applications via graph partitioning. In JeanM. Bacon and BrianF. Cooper, editors, *Middleware 2009*, volume 5896 of *Lecture Notes in Computer Science*, pages 308–327. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10444-2. Available from: http://dx.doi.org/10.1007/978-3-642-10445-9_16, doi:10.1007/978-3-642-10445-9_16.

[KhSHO11] Jinwoo Kim, Tae ho Shin, Soonhoi Ha, and Hyunok Oh. Resource minimized static mapping and dynamic scheduling of sdf graphs. In *ESTImedia*, pages 83–92, 2011.

[KK98] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, January 1998. ISSN 0743-7315. Available from: http://dx.doi.org/10.1006/jpdc.1997.1404, doi:10.1006/jpdc.1997.1404.

[KKK98] G. Karypis, V. Kumar, and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970. ISSN 1538-7305. doi:10.1002/j.1538-7305.1970.tb01770.x.

[KM66] Richard .M Karp and Raymond E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J. Applied Mathematics*, 14(4):1390–1411, 1966.

[KM08] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 114–124. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-860-2. Available from: http://doi.acm.org/10.1145/1375581.1375596, doi:10.1145/1375581.1375596.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997. ISBN 0792398947.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *Computers, IEEE Transactions on*, C-33(5):438–446, May 1984. ISSN 0018-9340. doi:10.1109/TC.1984.1676460.

[KRV06] Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 385–390. ACM, New York, NY, USA, 2006. ISBN 1-59593-134-1.

[KTA03] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 103–112. ACM, New York, NY, USA, 2003. ISBN 1-58113-647-1. doi:10.1145/780732.780747.

[Lan65] P. J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM*, 8(2):89–101, 1965. doi:10.1145/363744.363749.

[LDWL06] Shih-wei Liao, Zhaohui Du, Gansha Wu, and Guei-Yuan Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 196–207. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2499-0. doi:10.1109/CGO.2006.13.

[LFM04] Hu Liang, Meng Faner, and Hu Ming. A dynamic load balancing system based on data migration. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 1, pages 493–499 Vol.1, May 2004. doi:10.1109/CACWD.2004.1349072.

[Lit61] John D. C. Little. A Proof for the Queuing Formula: L= λ W. *Operations Research*, 9(3):383–387, 1961. ISSN 0030364X. doi:10.2307/167570.

[LLM88] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988. doi:10.1109/DCS.1988.12507.

[LM87a] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, 1987. ISSN 0018-9219. doi:10.1109/PROC.1987.13876.

[LM87b] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987. ISSN 0018-9340. doi:10.1109/TC.1987.5009446.

[Lo88] V.M. Lo. Heuristic algorithms for task assignment in distributed systems. *Computers, IEEE Transactions on*, 37(11):1384–1397, Nov 1988. ISSN 0018-9340. doi:10.1109/12.8704.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83:773–801, 1995.

[LR92] H.-C. Lin and C.S. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (lbc). *Software Engineering, IEEE Transactions on*, 18(2):148–158, Feb 1992. ISSN 0098-5589. doi:10.1109/32.121756.

[LR99] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, November 1999. ISSN 0004-5411. Available from: http://doi.acm.org/10.1145/331524.331526, doi:10.1145/331524.331526.

[LSK97] An-Chow Lai, Ce-Kuen Shieh, and Yih-Tzye Kok. Load balancing in distributed shared memory systems. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, pages 152 –158, feb 1997. doi:10.1109/PCCC.1997.581502.

[Mal09] Rajib Mall. *Real-Time Systems: Theory and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2009. ISBN 8131700690, 9788131700693.

[Mar66] David Frederic Martin. *Volume I. The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems. Volume Ii. Program Listings.* PhD thesis, 1966. AAI6606812.

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995. ISSN 0018-9162. Available from: http://dx.doi.org/10.1109/2.471178, doi:http://dx.doi.org/10.1109/2.471178.

[MCH+90] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: the second generation of a parallel program measurement system. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):206 –217, apr 1990. ISSN 1045-9219. doi:10.1109/71.80132.

[ME13] Changwoo Min and Young Ik Eom. DANBI: Dynamic scheduling of irregular stream programs for many-core systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 189–200. IEEE Press, Piscataway, NJ, USA, 2013. ISBN 978-1-4799-1021-2. Available from: http://dl.acm.org/citation.cfm?id=2523721.2523749.

[MG12] Avinash Malik and David Gregg. Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming. Technical report, School of Computer Science and Statistics, Trinity College Dublin, Ireland, February 2012. Available from: https://www.scss.tcd.ie/publications/tech-reports/tr-index.12.php.

[MG13] Avinash Malik and David Gregg. Orchestrating stream graphs using model checking. *TACO*, 10(3):19, 2013.

[MM08] Alexander V. Mirgorodskiy and Barton P. Miller. Diagnosing distributed systems with self-propelled instrumentation. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 82–103. Springer-Verlag New York, Inc., New York, NY, USA, 2008. ISBN 3-540-89855-7. Available from: http://dl.acm.org/citation.cfm?id=1496950.1496957.

[MMS09] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. Graph partitioning and disturbed diffusion. *Parallel Comput.*, 35(10-11):544–569, October 2009. ISSN 0167-8191. Available from: http://dx.doi.org/10.1016/j.parco.2009.09.006, doi:10.1016/j.parco.2009.09.006.

[MS04] Burkhard Monien and Stefan Schamberger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pages 198–205. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2240-8. doi:10.1109/SBAC-PAD.2004.18.

[MSA+85] J. Mcgraw, S. Skedzielewski, S. Allan, Oldehoeft Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2, 1985.

[MST+05] A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile. Performance technology for parallel and distributed component software. *Concurrency and Computation: Practice and Experience*, 17(2-4): 117–141, 2005. ISSN 1532-0634. Available from: http://dx.doi.org/10.1002/cpe.931, doi:10.1002/cpe.931.

[NSM+07] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen D. Malony, and Sameer Shende. TAUoverSupermon: Low-overhead online parallel performance monitoring. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 85–96. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-74465-7, 978-3-540-74465-8.

[PD10] Jongsoo Park and William J. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 1–10. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0079-7. Available from: http://doi.acm.org/10.1145/1810479.1810481, doi:10.1145/1810479.1810481.

[PG10] V. Petkov and M. Gerndt. Integrating parallel application development with performance analysis in Periscope. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010. doi:10.1109/IPDPSW.2010.5470940.

[PHG+10] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrière, and Eric Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2079 – 2088, 2010. ISSN 1877-0509. ICCS 2010. doi:DOI:10.1016/j.procs.2010.04.233.

[PHS+10] Frank Penczek, Stephan Herhut, Sven-Bodo Scholz, Alex Shafarenko, Jung-Sook Yang, Chun-Yi Chen, Nader Bagherzadeh, and Clemens Grelck. Message Driven Programming with S-Net: Methodology and Performance. *Parallel Processing Workshops, International Conference on*, 0:405–412, 2010.

ISSN 1530-2016. doi:http://doi.ieeecomputersociety.org/10.1109/ICPPW.2010.61.

[Pin08] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. ISBN 0387789340, 9780387789347.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-16488-3. Available from: http://dx.doi.org/10.1007/BFb0027047, doi:10.1007/BFb0027047.

[Pot97] Alex Pothen. Graph partitioning algorithms with applications to scientific computing. In *Parallel Numerical Algorithms*, pages 323–368. Kluwer Academic Press, 1997.

[PR96] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-61142-8_588.

[Pro10] Daniel Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.

[PSL90] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3): 430–452, May 1990. ISSN 0895-4798.

[PT91] G.M. Papadopoulos and K.R. Traub. Multithreading: a revisionist view of dataflow architectures. In *Computer Architecture, 1991. The 18th Annual International Symposium on*, pages 342–351, 1991.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74. ACM, New York, NY, USA, 1994. ISBN 0-89791-707-3. Available from: http://doi.acm.org/10.1145/192724.192731, doi:10.1145/192724.192731.

[RRB69] J. E. Rodrigues and Jorge E Rodriguez Bezos. A graph model for parallel computations. Technical report, Cambridge, MA, USA, 1969.

[RVK08] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. ISSN 1530-2075. doi:10.1109/IPDPS.2008.4536188.

[Sch04] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004. ISBN 3540002960. 12 pp.

[SK72] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Design Automation Workshop*, DAC '72, pages 57–62. ACM, New York, NY, USA, 1972. doi:10.1145/800153.804930.

[SK90] N.G. Shivaratri and P. Krueger. Two adaptive location policies for global scheduling algorithms. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 502 –509, may-1 jun 1990. doi:10.1109/ICDCS.1990.89320.

[SKH95] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. ISBN 0818665874.

[SKS92] N.G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33 –44, dec. 1992. ISSN 0018-9162. doi:10.1109/2.179115.

[SL93] G. C. Sih and E. A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):625–637, June 1993. ISSN 1045-9219. Available from: http://dx.doi.org/10.1109/71.242160, doi:10.1109/71.242160.

[SM00] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, August 2000. ISSN 0162-8828. doi:10.1109/34.868688.

[SM06] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006. ISSN 1094-3420. Available from: http://dx.doi.org/10.1177/1094342006064482, doi:10.1177/1094342006064482.

[SMM07] Sameer Shende, Allen D. Malony, and Alan Morris. Workload characterization using the tau performance system. In *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, PARA'06, pages 289–296. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-75754-6, 978-3-540-75754-2.

[SRU98] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: From dataflow to multithreading, 1998.

[SS92a] Sharon L. Smith and Robert B. Schnabel. Unstructured scientific computation on scalable multiprocessors. chapter Centralized and Distributed Dynamic Scheduling for Adaptive, Parallel Algorithms, pages 301–321. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-13272-9.

[SS92b] S.L. Smith and R.B. Schnabel. *Dynamic Scheduling Strategies for an Adaptive, Asynchronous Parallel Global Optimization Algorithm*. CU-CS. Department of Computer Science, University of Colorado, 1992. Available from: http://books.google.co.uk/books?id=jUu8SgAACAAJ.

[SS12] Peter Sanders and Christian Schulz. High quality graph partitioning. In *Graph Partitioning and Graph Clustering*, pages 1–18, 2012.

[Ste97] R. Stephens. A survey of stream processing. *Acta Informatica*, 34, 1997. ISSN 0001-5903.

[Sto77] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *Software Engineering, IEEE Transactions on*, SE-3(1):85–93, Jan 1977. ISSN 0098-5589. doi:10.1109/TSE.1977.233840.

[Sut66] William R. Sutherland. *The On-Line Graphical Specification of Computer Procedures*. PhD thesis, Massachusetts Institute of Technology, 1966.

[TDG+11] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 99–107. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21877-4. doi:10.1007/978-3-642-21878-1_13.

[TE92] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queuing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transaction on Automatic Control*, 37 (12):1936–1948, Dec. 1992.

[Thi09] William Thies. *Language and Compiler Support for Stream Programs*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009. Available from: http://groups.csail.mit.edu/commit/papers/09/thies-phd-thesis.pdf.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196. Springer-Verlag, London, UK, UK, 2002. ISBN 3-540-43369-4. Available from: http://dl.acm.org/citation.cfm?id=647478.727935.

[TKG+01] William Thies, Michal Karczmarek, Michael I. Gordon, David Z. Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. StreamIt: A compiler for streaming applications. Technical Report MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, Dec 2001. Available from: http://groups.csail.mit.edu/commit/papers/01/StreamIt-TM-622.pdf.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002. ISSN 0272-1732. Available from: http://dx.doi.org/10.1109/MM.2002.997877, doi:10.1109/MM.2002.997877.

[Trä06] Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan–Lin like heuristic. *Operations Research Letters*, 34(6):621 – 629, 2006. ISSN 0167-6377. doi:http://dx.doi.org/10.1016/j.orl.2005.10.003.

[UGT09a] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-0-7695-3576-0. Available from: http://dx.doi.org/10.1109/CGO.2009.20, doi:10.1109/CGO.2009.20.

[UGT09b] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '09,

pages 99–108. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-356-3. Available from: http://doi.acm.org/10.1145/1542452.1542466, doi:10.1145/1542452.1542466.

[WA05] Barry Wilkinson and Michael Allen. *Parallel programming - techniques and applications using networked workstations and parallel computers (2nd Edition)*. Pearson Education, 2005. ISBN 978-0-13-191865-8. 200-204 pp.

[WC00] C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, January 2000. ISSN 1064-8275. doi:10.1137/S1064827598337373.

[Wen75] K.S Weng. Stream oriented computation in recursive data-flow schemas. Technical report, Laboratory for Computer Science, MIT, Cambridge, MA, 1975.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980. ISSN 0001-0782. Available from: http://doi.acm.org/10.1145/358876.358882, doi:10.1145/358876.358882.

[WKPR05] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proc. 5th International Conference of Quality Software*. Melbourne, Australia, Sep. 2005.

[WO10] Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 307–318. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0178-7. Available from: http://doi.acm.org/10.1145/1854273.1854313, doi:10.1145/1854273.1854313.

[WP94] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994.

[XLM97] Zhichen Xu, James R. Larus, and Barton P. Miller. Shared-memory performance profiling. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 240–251. ACM, New York, NY, USA, 1997. ISBN 0-89791-906-8. Available from: http://doi.acm.org/10.1145/263764.263796, doi:10.1145/263764.263796.

[YM89] C.-Q. Yang and B.P. Miller. Performance measurement for parallel and distributed programs: a structured and automatic approach. *Software Engineering, IEEE Transactions on*, 15(12):1615 –1629, dec 1989. ISSN 0098-5589. doi:10.1109/32.58772.

[ZKS94] Y. Zhang, H. Kameda, and K. Shimizu. Adaptive bidding load balancing algorithms in heterogeneous distributed systems. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994., MASCOTS '94., Proceedings of the Second International Workshop on*, pages 250 –254, jan-2 feb 1994. doi:10.1109/MASCOT.1994.284414.

[ZLRA08] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008. ISSN 0163-5964. Available from: http://doi.acm.org/10.1145/1399972.1399978, doi:10.1145/1399972.1399978.