N° d'ordre : 518 *N° attribué par la bibliothèque : 07ENSL0518*

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme
et
- UNIVERSITÄT PASSAU -
Fakultät für Informatik und Mathematik

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

et

Doktor der Naturwissenschaften

au titre de l' École doctorale de Mathématiques et Informatique fondamentale

présentée et soutenue publiquement le 7 juillet 2009 par

Veronika REHN-SONIGO

Multi-criteria Mapping and Scheduling of Workflow Applications onto Heterogeneous Platforms

Directeur de thèse :	Yves	ROBERT	
Co-encadrants de thèse :	Anne	BENOIT	
	Harald	KOSCH	
Après avis de :	Umit	CATALYUREK	Rapporteur
	Michel	DAYDÉ	Rapporteur
	Harald	KOSCH	Rapporteur

Devant la commission d'examen formée de :

Anne	BENOIT	Membre
Umit	CATALYUREK	Rapporteur
Michel	DAYDÉ	Rapporteur
Harald	KOSCH	Rapporteur
Christian	LENGAUER	Membre
Yves	ROBERT	Membre
Denis	TRYSTRAM	Membre

DISSERTATION

Multi-criteria Mapping and Scheduling of Workflow Applications onto Heterogeneous Platforms

von
by

Veronika REHN-SONIGO

eingereicht in einem gemeinsamen Promostionsverfahren (Cotutelle) an der
submitted in cotutelle-procedure to the

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme
und an der
and to the

- UNIVERSITÄT PASSAU -
Fakultät für Informatik und Mathematik

zur Erlangung des Grades
in partial fulfillment of obtaining the degree

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

und zur Erlangung des Grades
and the degree

Doktor der Naturwissenschaften

Betreuer / Advisors :	Harald	KOSCH	Universität Passau
	Yves	ROBERT	ENS Lyon
Co-Advisor :	Anne	BENOIT	ENS Lyon
Reviewers :	Umit	CATALYUREK	Rapporteur
	Michel	DAYDÉ	Rapporteur
	Harald	KOSCH	Rapporteur

Remerciements

Je tiens tout d'abord à remercier mes rapporteurs pour leur travail.

Merci à Umit Catalyurek, Michel Daydé et Harald Kosch pour leur oeil critique sur mes travaux. Merci pour tous vos commentaires et propositions d'amélioration. La version actuelle de ma thèse a gagné, grâce à vous, en précision et lisibilité, et je vous en suis reconnaissante.

De même j'aimerais remercier les autres membres du jury.

Merci à Christian Lengauer et Denis Trystram d'avoir accepté de participer à mon jury de thèse et de s'être libérés pour venir à Lyon.

Sans oublier mes chers encadrants !

Merci à Anne, Harald et Yves, sans lesquels cette thèse n'aurait pas été la même. Merci pour votre soutien à tout moment, vos remarques critiques et vos idées innovatrices. J'ai passé un temps inoubliable avec vous.

Contents

Introduction **i**

I Replica Placement in Tree Networks **1**

1 Problem Definition **3**
1.1 Framework 4
1.1.1 Definitions and Notations 4
1.1.2 Problem Instances 5
1.2 Access Policies 7
1.2.1 Impact of the Access Policy on the Existence of a Solution 7
1.2.2 *Upwards* versus *Closest* 8
1.2.3 *Multiple* versus *Upwards* 9
1.2.4 Lower Bound for the Replica Counting Problem 9
1.3 Related Work 10

2 Replica Placement Strategies **13**
2.1 Complexity Results 13
2.1.1 With Homogeneous Nodes and the *Closest* Strategy 14
2.1.2 With Homogeneous Nodes and the *Multiple* Strategy 14
2.1.3 With Homogeneous Nodes and the *Upwards* Strategy 23
2.1.4 With Heterogeneous Nodes 24
2.2 Linear Programming Formulation 26
2.2.1 Single Server 26
2.2.2 Multiple Servers 27
2.2.3 A Mixed Integer LP-Based Lower Bound 28
2.3 Heuristics for the Replica Cost Problem 29
2.3.1 *Closest* 29
2.3.2 *Upwards* 30
2.3.3 *Multiple* 30
2.4 Experiments 30
2.4.1 Experimental Plan 31
2.4.2 Results 32

3 Multi-Criteria Optimization Problems **35**
3.1 Complexity Results 35
3.1.1 Replica Counting with QoS 35

3.1.2 Replica Cost with QoS and Bandwidth . . . 37
3.2 Linear Programming Formulation for Replica Placement with QoS . . . 44
3.2.1 Extension of the Mono-Criteria Linear Program . . . 45
3.2.2 An Exact MIP-Based Solution for *Multiple* . . . 45
3.3 Heuristics for the Replica Placement Problem with QoS Constraints . . . 47
3.3.1 *Closest* . . . 48
3.3.2 *Upwards* . . . 48
3.3.3 *Multiple* . . . 48
3.4 Experimental Plan . . . 49

II Pipeline Workflow Applications 55

4 Problem Definition 57
4.1 Framework . . . 58
4.1.1 Applicative Framework . . . 58
4.1.2 Target Platform . . . 59
4.1.3 Mapping Problem . . . 60
4.2 Motivating Examples . . . 62
4.3 Related Work . . . 63

5 Complexity Results 65
5.1 Mono-criteria Problems . . . 65
5.1.1 Failure Probability . . . 65
5.1.2 Latency . . . 66
5.1.3 Period . . . 67
5.2 Bi-criteria Optimization . . . 68
5.2.1 Period and Latency . . . 68
5.2.2 Latency and Failure Probability . . . 68
5.3 Linear Program Formulation . . . 72

6 Case Study 75
6.1 Principles of JPEG Encoding . . . 76
6.2 Heuristics . . . 76
6.2.1 Minimizing Latency for a Fixed Period . . . 76
6.2.2 Minimizing Period for a Fixed Latency . . . 77
6.3 Experimental Results . . . 77
6.3.1 General Experiments . . . 77
6.3.2 Experiments and Simulations for the JPEG encoder . . . 87

III Complex Streaming Applications 91

7 Introduction 93

8 In-Network Stream Processing 95
8.1 Models . . . 95
8.1.1 Application Model . . . 95
8.1.2 Platform Model . . . 96
8.1.3 Mapping Model and Constraints . . . 97
8.2 Complexity . . . 98
8.2.1 Linear Programming Formulation . . . 99
8.3 Heuristics . . . 102
8.4 Simulation Results . . . 104

9 Multiple Concurrent Applications 109
9.1 Framework . . . 109
9.1.1 Application Model . . . 109
9.1.2 Platform Model . . . 111
9.1.3 Mapping Model and Constraints . . . 111
9.1.4 Optimization Problems . . . 113
9.2 Complexity . . . 113
9.3 Linear Programming Formulation . . . 114
9.3.1 Input Data . . . 114
9.3.2 Variables . . . 115
9.3.3 Constraints . . . 115
9.3.4 Objective Function . . . 117
9.4 Heuristics . . . 118
9.5 Experimental Results . . . 120
9.5.1 Experimental Plan . . . 121
9.5.2 Results . . . 121

Conclusion and Perspectives 127
10.1 Conclusion . . . 127
10.1.1 Replica Placement . . . 127
10.1.2 Pipeline Workflow Applications . . . 127
10.1.3 Complex Streaming Applications . . . 128
10.2 Perspectives . . . 129
10.2.1 Extensions for the Replica Placement Problem . . . 129
10.2.2 Extensions for Workflow Applications . . . 130
10.2.3 Extensions for Concurrent Streaming Applications . . . 131
10.3 Final Remarks . . . 131

A Algorithms 1

B Bibliography 5

C Publications 11

D Notations 13

Introduction

Classical scheduling for parallel machines usually targets *makespan* minimization as unique objective. Given a set of tasks, the tasks are scheduled such that the last task finishes as early as possible. The total execution time, that is the time elapsed between the beginning and the end of processing of the entire set of jobs, is called the makespan. Jobs may have dependencies, and one speaks of a task graph that has to be scheduled. As the dependencies have to be respected, only some jobs can be scheduled, while others are beening processed. This considerably increases the difficulty of the scheduling problem. It is well known that simple instances of this classical optimization problem are already NP-hard even in the context of homogeneous resources [16].

These days, new technologies have been developed and computing resources have become more accessible. Consequently, new platform structures have arisen, and we now have to deal with large-scale heterogeneous platforms: a large number of different speed processors, interconnected via communication links, is available for computation purposes. A recent form of large-scale platform utilization is called cloud computing, where some well-chosen processing units are rent for computation. Of course these new technologies also bring new parameters to the scheduler. For example platform sizes have dramatically increased: heterogeneous clusters of several hundreds or thousands of processors are not uncommon. It is only natural that with such large orders of magnitude, processors come to crash down more likely while some applications may actually be running. Consequently, one of the new important parameters arising for evaluation is *reliability*.

Also the application *throughput* becomes an important factor. Dealing with workflows, you are not only interested in the computation of one data set, but rather of hundreds of data sets. It becomes an important criteria to determine how many data sets can be processed in a certain time interval. Thus, simple makespan minimization is not sufficient anymore and one has to cope with a large variety of new constraints: one may think of platform costs, quality of service, energy consumption, reliability and so on.

Another recent trend, which is a natural consequence of the new optimization parameters, is dealing with *multi-criteria* objectives. Here we aim at combining two or more parameters at once in the optimization objective. For example we try to map an application on a set of processors so as to achieve a good throughput but at the same time a high reliability. A crucial new difficulty arises: how to combine objectives? There are two main tendencies. The first combines all objectives into a single objective function and optimizes them simultaneously. This leads to so called Pareto optimality, where an entire solution space is spanned. The other tendency fixes one or more parameters beforehand and optimizes the last parameter. The motivation for this approach is the following: frequently the different parameters are antagonist and not comparable. Fixing one parameter, one accepts a certain threshold for this parameter, which enforces a constraint to get the best solution for another. In the previous example, we would impose a throughput of, say, 100 frames per second, and then look for the most reliable solution

(which seems more natural than combining both parameters).

This thesis is situated in the domain of multi-criteria optimization problems for large-scale heterogeneous platforms. More precisely we focus on streaming applications. In this work, we consider three types of applications. The first type concerns applications like video on demand (VOD), where requests are processed in a hierarchical network. The second type are linear workflow applications, which include image processing applications like the JPEG encoder. Third, we consider tree-shaped applications that occur in video surveillance or relational databases. All these applications offer multiple points for optimization and we opt for the optimization technique where we fix one or several parameters to optimize the last one. Throughout this thesis, we aim of characterizing the different mapping problems, and to assess their complexity from a theoretical point of view.

In the following we detail the structure and contributions of this work. The thesis is split into three main parts, according to the different application types, and to the corresponding scheduling and mapping problems.

Replica Placement in Tree Networks

The first part is based on the applicative problems of VOD. The typical platform has hierarchical shape. We discuss and compare several policies to place replicas in tree networks, subject to server capacity, quality of service (QoS) and bandwidth constraints. The client requests are known beforehand, while the number and location of the servers are to be determined. We give an introduction to the subject in Chapter 1. In Section 1.1 we introduce the framework with definitions and notations. The standard approach in the literature is to enforce that all requests of a client be served by the closest server in the tree and we refer to this approach as *Closest* policy. We introduce two new policies in Section 1.2. In the first policy, *Upwards*, all requests from a given client are still processed by the same server, but this server can be located anywhere in the path from the client to the root. In the second policy, the requests of a given client can be processed by multiple servers and we name this policy *Multiple*. An overview of related work is presented in Section 1.3. One major contribution of this part is to assess the impact of these new policies on the total replication cost. Another important goal is to assess the impact of server heterogeneity, both from a theoretical and a practical perspective. We study this mono-criterion approach in Chapter 2. *Upwards* is NP-complete in the homogeneous case, whereas *Multiple* can be solved in polynomial time. This latter result is quite unexpected, and we provide an elegant algorithm to compute the optimal cost for this policy. Not surprisingly, all three policies turn out to be NP-complete for heterogeneous platforms. We detail these complexity results in Section 2.1.

In Chapter 3 we study the influence of QoS and bandwidth constraints on the solution. *Multiple* with homogeneous nodes and QoS constraints becomes NP-complete (Section 3.1). But we still provide a polynomial-time algorithm to solve *Closest* with QoS and bandwidth constraints on this type of platform. In Sections 2.2 and 3.2, we formulate all problems as linear programs.

On the practical side we design several heuristics, both for the mono-criterion optimization (Section 2.3) as well as for the problem with QoS constraints (Section 3.3). In our experiments we are able to emphasize the importance of the new access policies, as the number of solutions (and their quality) increases using *Upwards* or even better *Multiple*. Owing to the linear programs, we are able to assess the absolute performance of our heuristics in Sections 2.4 and 3.4.

This work has been published in [A2, B5, B6, B7].

Pipeline Workflow Applications

In the second part we focus on linear workflow applications. A typical application class is digital image coding, where images are processed in steady-state mode. These applications can be expressed as pipeline graphs where a series of data sets (tasks) enter the input stage and progress from stage to stage until the final result is computed. Each stage has its own communication and computation requirements: it reads an input file from the previous stage, processes the data and outputs a result to the next stage. For each data set, initial data is input to the first stage, and final results are output from the last stage. We introduce the precise problem in Chapter 4.

Chapter 5 presents our complexity results. We focus on three antagonist optimization criteria and their combination: the period, i.e., the inverse of the application throughput; the latency, i.e., the response time; and the reliability, i.e., the probability that the computation will be successful. Throughput can be minimized by cutting the pipeline into several parts in order to increase parallelism. Latency is minimized by using faster processors, while reliability is increased by replicating computations on a set of processors. However, replication increases latency (additional communications, slower processors). The application fails to be executed only if all the replicated processors fail during execution.

In Section 5.1, we concentrate on mono-criteria optimization and we assess the problem complexity of all three criteria on different platform types. We then tackle the bi-criteria optimization in Section 5.2. The combination of period-latency is NP-complete on platforms with different speed processors, as period minimization already is NP-complete. Focusing on the combination of reliability and latency, the problem is polynomial for *Fully Homogeneous*, NP-hard for *Fully Heterogeneous* and remains an open problem for *Communication Homogeneous*.

Furthermore we examine a particular pipeline workflow application in a case study (Chapter 6): the JPEG encoder pipeline. We chose this application as it can be used with workflows, when encoding M-JPEG video streams. We provide several polynomial heuristics (Section 6.2 and we evaluate their behavior for period-latency optimization in Section 6.3. Our experiments point out the correlation between efficiency and platform parameters.

This work has been published in [A1, B2, B3, B4].

Query Streaming

The third part tackles the operator mapping problem for in-network stream processing. This problem is kind of a combination of the previous two. We investigate the problem of request dissemination in a tree of operators. The operators have to be mapped onto processors to compute some data. At the same time, the processors perform downloads from servers to update their data. The goal is to produce some final results at some desired rate. The latter requirement can be considered as a QoS constraint. In practice, the execution of the operators on the data stream is distributed over the network. Hence we have to determine where to perform the computation of each operator such that the fixed application rate (QoS request) is respected. Examples of in-network stream processing include the processing of data in a sensor network, or of continuous queries on distributed relational databases.

In Chapter 8, we study the problem in a constructive scenario. Our aim is to provide the user with a set of processors that should be bought or rented in order to ensure that the application

achieves a minimum steady-state throughput, and with the objective of minimizing platform cost. The problem turns out to be NP-hard even for the simplest instances (Section 8.2) and we formulate the problem as integer linear program in Section 8.2.1. Furthermore we design a set of mapping heuristics in Section 8.3. In Section 8.4 we evaluate the absolute performance of these heuristics via extensive simulation and in comparison to the optimal linear program solution. We are able to identify one heuristic which almost always produces optimal results and almost always outperforms the other heuristics.

A follow-on of this work is the subject of Chapter 9, namely the extension to concurrent applications. Instead of one single application, we consider in this chapter a set of applications that have to be executed simultaneously in the network. Each application has its own throughput requirement, i.e., its own QoS request. In Chapter 9, we abandon the constructive framework and we deploy the applications on an existing target platform. This scenario is more realistic: with one application for a single user, this user wants to build a platform dedicated to his/her needs; on the contrary, with several applications from several users running concurrently, it is more likely to share an existing set of resources for a common deployment. To improve performance, different operator trees may reuse common sub-expressions when operator trees share the same subtrees. The framework is detailed in Section 9.1. We propose several different optimization problems. Their complexity analysis, presented in Section 9.2, classifies all of them as NP-hard. Another contribution is a set of polynomial heuristics, that is presented in Section 9.4. We designed them for one of the optimization problems that we study more in detail. The experiments in Section 9.5 consolidate the importance of the reuse of common subexpression for the performance.

Part of this work has been published in [B1].

Part I

Replica Placement in Tree Networks

Chapter 1

Problem Definition

This chapter deals with the general problem of replica placement in tree networks. Informally, there are clients issuing requests to be satisfied by servers. The clients are known (both their position in the tree and their number of requests), while the number and location of the servers are to be determined. A client is a leaf node of the tree, and its requests can be served by one or several internal nodes. Initially, there are no replica; when a node is equipped with a replica, it can process a number of requests, up to its capacity limit. Nodes equipped with a replica, also called servers, can only serve clients located in their subtree (so that the root, if equipped with a replica, can serve any client); this restriction is usually adopted to enforce the hierarchical nature of the target application platforms, where a node has knowledge only of its parent and children in the tree.

The rule of the game is to assign replicas to nodes so that some optimization function is minimized. Typically, this optimization function is the total utilization cost of the servers. If all the nodes are identical, this reduces to minimizing the number of replicas. If the nodes are heterogeneous, it is natural to assign a cost proportional to their capacity (so that one replica on a node capable of handling 200 requests is equivalent to two replicas on nodes of capacity 100 each). We call this optimization problem Replica Placement in the following.

In mono-criterion placement (Cf. Chapter 2), we focus on optimizing the total utilization cost (or replica number in the homogeneous case). Chapter 3 whereas deals with the Replica Placement as multi-criteria optimization problem. Additional constraints are introduced, in order to guarantee some Quality of Service (QoS): requests must be served in limited time, thereby prohibiting remote or hard-to-reach replica locations. Also, the flow of requests through a link in the tree cannot exceed some bandwidth-related capacity.

We point out that the distribution tree (clients and nodes) is fixed in our approach. This key assumption is quite natural for a broad spectrum of applications, such as electronic, ISP, or VOD service delivery [39, 22, 74, 46]. The root server has the original copy of the database but cannot serve all clients directly, so a distribution tree is deployed to provide a hierarchical and distributed access to replicas of the original data. On the contrary, in other, more decentralized, applications (e.g., allocating Web mirrors in distributed networks), a two-step approach is used: first determine a “good” distribution tree in an arbitrary interconnection graph, and then determine a “good” placement of replicas among the tree nodes. Both steps are interdependent, and the problem is much more complex, due to the combinatorial solution space (the number of candidate distribution trees may well be exponential).

In most papers from the literature (see Section 1.3 for a survey of related work), all requests of a client are served by the closest replica, i.e., the first replica found in the unique path from

the client to the root in the distribution tree. This *Closest* policy is simple and natural, but may be unduly restrictive, leading to a waste of resources. We introduce two different approaches:

In the first one, we keep the restriction that all requests from a given client are processed by the same replica, but we allow client requests to "traverse" servers so as to be processed by other replicas located higher in the path (closer to the root). We call this approach the *Upwards* policy. The trade-off to explore is the following: the *Closest* policy assigns replicas at proximity of the clients, but may need to allocate too many of them if some local subtree issues a great number of requests. The *Upwards* policy will ensure a better resource usage, load-balancing the process of requests on a larger scale; the possible drawback is that requests will be served by remote servers, likely to take longer time to process them. Taking QoS constraints into account would typically be more important for the *Upwards* policy.

In the second approach, we further relax access constraints and grant the possibility for a client to be assigned several replicas. With this *Multiple* policy, the processing of a given client's requests will be split among several servers located in the tree path from the client to the root. Obviously, this policy is the most flexible, and likely to achieve the best resource usage. The only drawback is the (modest) additional complexity induced by the fact that requests must now be tagged with the replica server ID in addition to the client ID.

Our major contributions are as follows: i) the comparison of the three access policies, *Closest*, *Upwards* and *Multiple*; ii) we assess the impact of server heterogeneity, both from a theoretical and a practical perspective; iii) we evaluate the influence of QoS and bandwidth constraints on the solution.

1.1 Framework

This section is devoted to a precise statement of the Replica Placement optimization problem. We start with some definitions and notations. Next we outline the simplest instance of the problem. Then we describe several types of constraints that can be added to the formulation.

1.1.1 Definitions and Notations

We consider a distribution tree $\mathcal{T}$ whose nodes are partitioned into a set of clients $\mathcal{C}$ and a set of nodes $\mathcal{N}$. The set of tree edges is denoted as $\mathcal{L}$. The clients are leaf nodes of the tree, while $\mathcal{N}$ is the set of internal nodes. It would be easy to allow *client-server* nodes which play both the rule of a client and of an internal node (possibly a server), by dividing such a node into two distinct nodes in the tree, connected by an edge with zero communication cost.

A *client* $i \in \mathcal{C}$ is making r_i requests per time unit to a database. For the sake of clarity, we restrict the presentation to a single object type, hence a single database. We deal with several object types in Section 10.2.1.

A *node* $j \in \mathcal{N}$ may or may not have been provided with a replica of the database. Nodes equipped with a replica (i.e., servers) can process up to W_j requests per time unit from clients in their subtree. In other words, there is a unique path from a client i to the root of the tree, and each node in this path is eligible to process some or all the requests issued by i when provided with a replica. We denote by $\mathsf{Servers}(i) \subseteq \mathcal{N}$ this set of nodes. The price to pay to place a replica at node j is sc_j.

Let r be the root of the tree. If $j \in \mathcal{N}$, then $\mathsf{children}(j)$ is the set of children of node j. If $k \neq r$ is any node in the tree (leaf or internal), $\mathsf{parent}(k)$ is its parent in the tree. If $l : k \to k' = \mathsf{parent}(k)$ is any link in the tree, then $\mathsf{succ}(l)$ is the link $k' \to \mathsf{parent}(k')$ (when it exists). Let

$\mathsf{Ancestors}(k)$ denote the set of ancestors of node k, i.e., the nodes in the unique path that leads from k up to the root r (k excluded). If $k' \in \mathsf{Ancestors}(k)$, then $\mathsf{path}[k \to k']$ denotes the set of links in the path from k to k'; also, $\mathsf{subtree}(k)$ is the subtree rooted in k, including k.

We introduce more notations to describe our system in the following.

- **Clients** $i \in \mathcal{C}$ – Each client i (leaf of the tree) is sending r_i requests per time unit. For such requests, the required QoS (typically, a response time) is denoted q_i, and we need to ensure that this QoS will be satisfied for each client.

- **Nodes** $j \in \mathcal{N}$ – Each node j (internal node of the tree) has a processing capacity W_j, which is the total number of requests that it can process per time-unit when it has a replica. A cost is also associated to each node, sc_j, which represents the price to pay to place a replica at this node. With a single object type it is quite natural to assume that sc_j is proportional to W_j: the more powerful a server, the more costly. But with several objects we may use non-related values of capacity and cost.

- **Communication links** $l \in \mathcal{L}$ – The edges of the tree represent the communication links between nodes (leaf and internal). We assign a communication time comm_l on link l which is the time required to send a request through the link.

1.1.2 Problem Instances

For each client $i \in \mathcal{C}$, let $\mathsf{Servers}(i) \subseteq \mathcal{N}$ be the set of servers responsible for processing at least one of its requests. We do not specify here which access policy is enforced (e.g. one or multiple servers), we defer this to Section 1.2. Instead, we let $r_{i,j}$ be the number of requests from client i processed by server j (of course, $\sum_{j \in \mathsf{Servers}(i)} r_{i,j} = r_i$). In the following, R is the set of replicas:

$$R = \{j \in \mathcal{N} | \ \exists i \in C \ , \ j \in \mathsf{Servers}(i)\} \, .$$

Constraints

Three main types of constraints are considered.

Server capacity – The problem is constrained by the fact that no server capacity can be exceeded is present in all variants of the problem:

$$\forall j \in R, \sum_{i \in \mathcal{C} | j \in \mathsf{Servers}(i)} r_{i,j} \leq \mathsf{W}_j .$$

QoS – Some problem instances enforce a quality of service: the time to transfer a request from a client to a replica server is bounded by a quantity q_i. This translates into:

$$\forall i \in \mathcal{C}, \forall s \in \mathsf{Servers}(i), \sum_{l \in \mathsf{path}[i \to s]} \mathsf{comm}_l \leq \mathsf{q}_i .$$

Note that it would be easy extend the QoS constraint so as to take the computation cost of a request in addition to its communication cost. This former cost is directly related to the computational speed of the server and the amount of computation (in flops) in required for each request.

Link capacity – Some problem instances enforce a global constraint on each communication link $l \in \mathcal{L}$:

$$\sum_{i \in \mathcal{C}, s \in \mathsf{Servers}(i) | l \in \mathsf{path}[i \to s]} r_{i,s} \leq \mathsf{BW}_l.$$

In the multiple servers access policy with multiple objects, $r_{i,s}^{(k)}$ represents the amount of requests from client i processed by server s on object k. The single server case is obtained by replacing $\mathsf{Servers}(i,k)$ by $\mathsf{server}(i,k)$ (the single server processing requests on k for client i), and the $r_{i,s}^{(k)}$ are simplified to $r_i^{(k)}$.

Objective Function

The objective function for the Replica Placement problem is defined as:

$$\text{Min} \sum_{s \in R} \mathsf{sc}_s.$$

As already pointed out, it is frequently assumed that the cost of a server is proportional to its capacity, so in some problem instances we let $\mathsf{sc}_s = \mathsf{W}_s$.

Simplified Problems

We define a few simplified problem instances in the following, that can be divided into mono-criteria objective and multi-criteria objective:

a) Mono-criterion optimization

Only server capacities – The problem without QoS and link capacities reduces to finding a valid solution of minimal cost, where "valid" means that no server capacity is exceeded. We name this fundamental problem as Replica Cost.

Replica counting – We can further simplify the previous Replica Cost problem in the homogeneous case: with identical servers, the Replica Cost problem amounts to minimize the number of replicas needed to solve the problem. In this case, the storage cost sc_j is set to 1 for each node. We call this problem Replica Counting.

b) Optimizing multiple criteria

QoS=distance – We can simplify the expression of the communication time in the QoS constraint and only consider the distance (in number of hops) between a client and its server(s). The QoS constraint is then

$$\forall i \in \mathcal{C}, \forall s \in \mathsf{Servers}(i),\ d(i,s) \leq \mathsf{q}_i$$

where the distance $d(i,s) = |\mathsf{path}[i \to s]|$ is the number of communication links between i and s.

No link capacity – We may consider the problem assuming infinite link capacity, i.e., not bounding the total traffic on any link in an admissible solution.

We name Replica Cost with QoS the bi-criteria problem that simplifies Replica Placement to QoS=distance without link capacity. Its homogeneous counterpart is called Replica Counting with QoS.

1.2 Access Policies

In this section we review the usual policies enforcing which replica is accessed by a given client. Consider that each client i is making r_i requests per time-unit. There are two scenarios for the number of servers assigned to each client:

Single server – Each client i is assigned a single server $\mathsf{server}(i)$, that is responsible for processing all its requests.

Multiple servers – A client i may be assigned several servers in a set $\mathsf{Servers}(i)$. Each server $s \in \mathsf{Servers}(i)$ will handle a fraction $r_{i,s}$ of the requests. Of course $\sum_{s\in\mathsf{Servers}(i)} r_{i,s} = r_i$.

To the best of our knowledge, the single server policy has been enforced in all previous approaches. One objective of this work is to assess the impact of this restriction on the performance of data replication algorithms. The single server policy may prove a useful simplification, but may come at the price of a non-optimal resource usage.

In the literature, the single server strategy is further constrained to the *Closest* policy. Here, the server of client i is constrained to be the first server found on the path that goes from i upwards to the root of the tree. In particular, consider a client i and its server $\mathsf{server}(i)$. Then any other client node i' residing in the subtree rooted in $\mathsf{server}(i)$ will be assigned a server in that subtree. This forbids requests from i' to "traverse" $\mathsf{server}(i)$ and be served higher (closer to the root in the tree).

We relax this constraint in the *Upwards* policy which is the general single server policy. Notice that a solution to *Closest* always is a solution to *Upwards*, thus *Upwards* is always better than *Closest* in terms of the objective function. Similarly, the *Multiple* policy is always better than *Upwards*, because it is not constrained by the single server restriction.

The following sections illustrate the three policies. Section 1.2.1 provides simple examples where there is a valid solution for a given policy, but none for a more constrained one. Section 1.2.2 shows that *Upwards* can be arbitrarily better than *Closest*, while Section 1.2.3 shows that *Multiple* can be arbitrarily better than *Upwards*. We conclude with an example showing that the cost of an optimal solution of the REPLICA COUNTING problem (for any policy) can be arbitrarily higher than the obvious lower bound

$$\left\lceil \frac{\sum_{i\in\mathcal{C}} r_i}{W} \right\rceil ,$$

where W is the server capacity.

1.2.1 Impact of the Access Policy on the Existence of a Solution

We consider here a very simple instance of the REPLICA COUNTING problem. In this example there are two nodes, s_1 being the unique child of s_2, the tree root (see Figure 1.1). Each node can process $\mathsf{W} = 1$ request.

- If s_1 has one client child making 1 request, the problem has a solution with all three policies, placing a replica on s_1 or on s_2 indifferently (Figure 1.1(a)).
- If s_1 has two client children, each making 1 request, the problem has no more solution with *Closest*. However, we have a solution with both *Upwards* and *Multiple* if we place replicas on both nodes. Each server will process the request of one of the clients (Figure 1.1(b)).

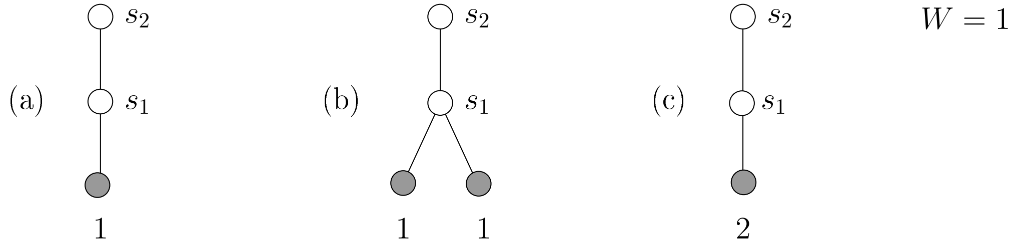


Figure 1.1: Access policies.

- Finally, if s_1 has only one client child making 2 requests, only *Multiple* has a solution since we need to process one request on s_1 and the other on s_2, thus requesting multiple servers (Figure 1.1(c)).

This example demonstrates the usefulness of the new policies. The *Upwards* policy allows to find solutions when the classical *Closest* policy does not. The same holds true for *Multiple* versus *Upwards*. In the following, we compare the cost of solutions obtained with different strategies.

1.2.2 *Upwards* versus *Closest*

In the following example, we construct an instance of Replica Counting where the cost of the *Upwards* policy is arbitrarily lower than the cost of the *Closest* policy. We consider the tree network of Figure 1.2, where there are $2n+2$ internal nodes, each with $\mathsf{W}_j = W = n$, and $2n+1$ clients, each with $r_i = r = 1$.

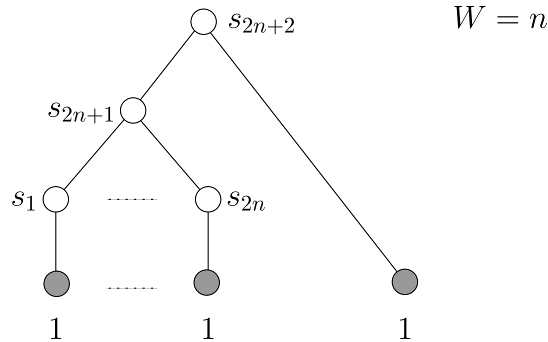


Figure 1.2: *Upwards* versus *Closest*

With the *Upwards* policy, we place three replicas in s_{2n}, s_{2n+1} and s_{2n+2}. All requests can be satisfied with these three replicas.

When considering the *Closest* policy, first we need to place a replica in s_{2n+2} to cover its client. Then,

- Either we place a replica on s_{2n+1}. In this case, this replica is handling n requests, but there remain n other requests from the $2n$ clients in its subtree that cannot be processed by s_{2n+2}. Thus, we need to add n replicas between $s_1..s_{2n}$.

- Otherwise, $n-1$ requests of the $2n$ clients in the subtree of s_{2n+1} can be processed by s_{2n+2} in addition to its own client. We need to add $n+1$ extra replicas among $s_1, s_2, \dots, s_{2n}$.

In both cases, we are placing $n+2$ replicas, instead of the 3 replicas needed with the *Upwards* policy. This proves that *Upwards* can be arbitrary better than *Closest* on some REPLICA COUNTING instances.

1.2.3 *Multiple* versus *Upwards*

In this section we build an instance of the REPLICA COUNTING problem where *Multiple* is arbitrarily better than *Upwards*.

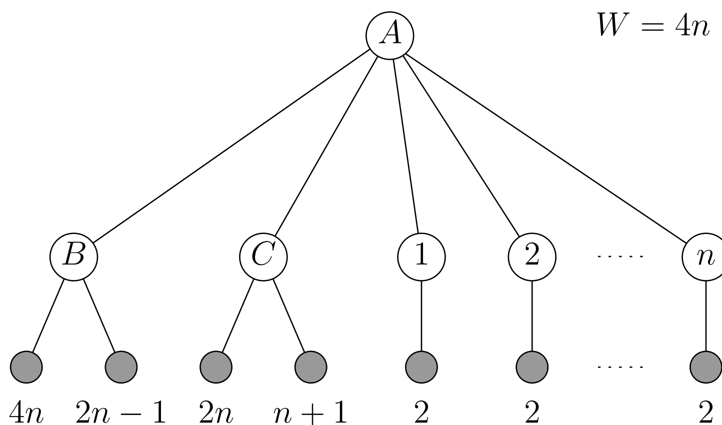


Figure 1.3: *Multiple* versus *Upwards*, homogeneous platforms.

Consider the instance of REPLICA COUNTING represented in Fig. 1.3, with $3+n$ nodes of capacity $\mathsf{W} = 4n$. The root A has $n+2$ children nodes B, C and $1, ..., n$. Node B has two client children, one with $4n$ requests and the other with $2n-1$ requests. Node C has two client children, one with $2n$ requests and the other with $2n+1$ requests. Each node numbered i has a unique child, a client with 2 requests.

The *Multiple* policy assigns 3 replicas to A, B and C. B handles the $4n$ requests of its first client, while the other client is served by A. C handles $2n$ requests from both of its clients, and the 1 remaining request is processed by A. Server A therefore processes $(2n-1)+1 = 2n$ requests coming up from B and C. Requests coming from the n remaining nodes sum up to $2n$, thus A is able to process all of them.

For the *Upwards* policy, we need to assign replicas everywhere. Indeed, with this policy, C cannot handle more than $2n+1$ requests since it is unable to process requests from both of its children, and thus A has $(2n-1)+2n$ requests coming from B and C. It cannot handle any of the $2n$ remaining requests, and thus each remaining node must process requests coming from its own client. This leads to a total of $n+3$ replicas.

The performance factor is thus $\frac{n+3}{3}$, which can be arbitrarily big when n becomes large.

1.2.4 Lower Bound for the Replica Counting Problem

Obviously, the cost of an optimal solution of the REPLICA COUNTING problem (for any policy) cannot be lower than the obvious lower bound $\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil$, where W is the server capacity. Indeed,

this corresponds to a solution where the total request load is shared as evenly as possible among the replicas.

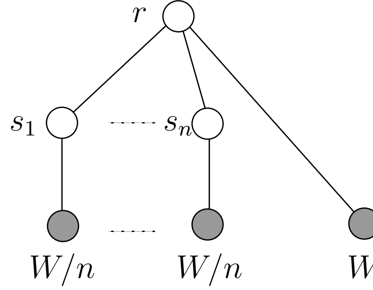


Figure 1.4: The lower bound cannot be approximated for Replica Counting.

The following instance of Replica Counting shows that the optimal cost can be arbitrarily higher than this lower bound. Consider Figure 1.4, with $n+1$ nodes of capacity $\mathsf{W}_j = W$. The root r has $n+1$ children, n nodes labeled s_1 to s_n, and a client with $r_i = W$. Each node s_j has a unique child, a client with $r_i = W/n$ (assume without loss of generality that W is divisible by n). The lower bound is $\left\lceil \frac{\sum_{i \in \mathcal{C}} r_i}{W} \right\rceil = \frac{2W}{W} = 2$. However, each of the three policies *Closest*, *Upwards* and *Multiple* will assign a replica to the root to cover its client, and will then need n extra replicas, one per client of s_j, $1 \leq j \leq n$. The total cost is thus $n+1$ replicas, arbitrarily higher than the lower bound.

All the examples in Sections 1.2.1 to 1.2.4 give an insight of the combinatorial nature of the Replica Placement optimization problem, even in its simplest variants Replica Cost and Replica Counting.

1.3 Related Work

Early work on replica placement by Wolfson and Milo [76] has shown the impact of the write cost and motivated the use of a minimum spanning tree to perform updates between the replicas. In this work, they prove that the replica placement problem in a general graph is NP-complete, even without taking into account storage costs. Thus they address the case of special topologies, and in particular tree networks. They give a polynomial solution in a fully homogeneous case and a simple model with no QoS and no server capacity. Their work uses the closest server access policy (single server) to access the data.

Using this *Closest* policy, Cidon et al. [22] studied an instance of the problem with multiple objects. In this work, the objective function has no update cost, but integrates a communication cost. Communication cost in the objective function can be seen as a substitute for QoS. Thus, they minimize the average communication cost for all the clients rather than ensuring a given QoS for each client. They target fully homogeneous platforms since there are no server capacity constraints in their approach. A similar instance of the problem has been studied by Liu et al. [46], adding a QoS in terms of a range limit, and the objective being the Replica Counting problem. In this latter approach, the servers are homogeneous, and their capacity is bounded.

Cidon et al. [22] and Liu et al. [46] both use the *Closest* access policy. In each case, the optimization problems are shown to have polynomial complexity. However, the variant with bidirectional links is shown NP-complete by Kalpakis et al. [39]. Indeed in [39], requests can be

served by any node in the tree, not just the nodes located in the path from the client to the root. The simple problem of minimizing the number of replicas with identical servers of fixed capacity, without any communication cost nor QoS constraints, directly reduces to the classical bin packing problem.

Kalpakis et al. [39] show that a special instance of the problem is polynomial, when considering no server capacities, but with a general objective function taking into account read, write and storage costs. In their work, a minimum spanning tree is used to propagate the writes, as was done in [76]. Different methods can however be used, such as a minimum cost Steiner tree, in order to further optimize the write strategy [40].

All papers listed above consider the *Closest* access policy. As already stated, most problems are NP-complete, except for some very simplified instances. Karlsson et al. [42, 41] compare different objective functions and several heuristics to solve these complex problems. They do not take QoS constraints into account, but instead integrate a communication cost in the objective function as was done in [22]. Integrating the communication cost into the objective function can be viewed as a Lagrangian relaxation of QoS constraints.

Tang and Xu [68] have been one of the first authors to introduce actual QoS constraints in the problem formalization. In their approach, the QoS corresponds to the latency requirements of each client. Different access policies are considered. First, a replica-aware policy in a general graph is proven to be NP-complete. When the clients do not know where the replicas are (replica-blind policy), the graph is simplified to a tree (fixed routing scheme) with the *Closest* policy, and in this case again it is possible to find a polynomial algorithm using dynamic programming.

To the best of our knowledge, there is no related work comparing different access policies, either on tree networks or on general graphs. Most previous works impose the *Closest* policy. The *Multiple* policy is enforced by Rodolakis et al. [58] but in a very different context. In fact, they consider general graphs instead of trees, so they face the combinatorial complexity of finding good routing paths. Also, they assume an unlimited capacity at each node, since they can add numerous servers of different kinds on a single node. Finally, they include some QoS constraints in their problem formulation, based on the round trip time (in the graph) required to serve the client requests. In such a context, this (very particular) instance of the *Multiple* problem is shown to be NP-hard.

Chapter 2

Replica Placement Strategies

In this chapter we deal with the mono-criterion optimization of Replica Placement, where we exclusively deal with server capacities.

One major contribution of this chapter is to assess the impact of server heterogeneity, both from a theoretical and a practical perspective. In Section 2.1 we establish several complexity results. Section 2.2 deals with the formulation for the Replica Placement problem in terms of an integer linear program. In Section 2.3 we introduce several polynomial heuristics to solve the Replica Placement problem with the different access policies. These heuristics are compared through simulations, whose results are analyzed in Section 2.4.

2.1 Complexity Results

In this section we aim at assessing the impact of the access policy on the problem with homogeneous versus heterogeneous servers. We consider a tree $\mathcal{T} = \mathcal{C} \cup \mathcal{N}$. Each client $i \in \mathcal{C}$ has r_i requests; each node $j \in \mathcal{N}$ has processing capacity W_j and storage cost $\mathsf{sc}_j = W_j$. The problem comes in two flavors, either the Replica Counting problem with homogeneous nodes ($\mathsf{W}_j = \mathsf{W}$ for all $j \in \mathcal{N}$), or the Replica Cost problem with heterogeneous nodes (servers with different capacities/costs).

In the single server version of the problem, we need to find a server $\mathsf{server}(i)$ for each client $i \in \mathcal{C}$. R is the set of replica, i.e., the servers chosen among the nodes in $\mathcal{N}$. The only constraint is that server capacities cannot be exceeded: this translates into

$$\sum_{i \in \mathcal{C}, \mathsf{server}(i)=j} r_i \leq W_j \quad \text{for all } j \in \mathcal{N}.$$

The objective is to find a valid solution of minimal storage cost $\sum_{j \in R} W_j$. Note that with homogeneous nodes, the problem reduces to find the minimum number of servers, i.e., to the Replica Counting problem. As outlined in Section 1.2, there are two variants of the single server version of the problem, namely the *Closest* and the *Upwards* strategies.

In the *Multiple* policy with multiple servers per client, for any client $i \in \mathcal{C}$ and any node $j \in \mathcal{N}$, $r_{i,j}$ is the number of requests from i that are processed by j ($r_{i,j} = 0$ if $j \notin R$, and $\sum_{j \in \mathcal{N}} r_{i,j} = r_i$ for all $i \in \mathcal{C}$). The capacity constraint now writes

$$\sum_{i \in \mathcal{C}} r_{i,j} \leq W_j \quad \text{for all } j \in R,$$

	Homogeneous (REPLICA COUNTING)	**Heterogeneous** (REPLICA COST)
Closest	polynomial [22, 46]	NP-hard
Upwards	NP-hard	NP-hard
Multiple	polynomial	NP-hard

Table 2.1: Complexity results for the different instances of the problem.

while the objective function is the same as for the single server version.

The decision problems associated with the previous optimization problems are easy to formulate: given a bound on the number of servers (homogeneous version) or on the total storage cost (heterogeneous version), is there a valid solution that meets the bound?

Table 2.1 captures the complexity results. The NP-completeness of the *Upwards*/Homogeneous case comes as a surprise, since all previously known instances were shown to be polynomial, using dynamic programming algorithms. In particular, the *Closest*/Homogeneous variant remains polynomial when adding communication costs [22] or QoS constraints [46], Cf. Chapter 3. We provide an elegant algorithm to show the polynomial complexity of the *Multiple*/Homogeneous problem.

Previous NP-completeness results involved general graphs rather than trees, and the combinatorial nature of the problem came from the difficulty to extract a good replica tree out of an arbitrary communication graph. Here the tree is fixed, but the problem remains combinatorial due to resource heterogeneity.

2.1.1 With Homogeneous Nodes and the *Closest* Strategy

Cidon at al. [22] proved that REPLICA COUNTING with the *Closest* strategy on homogeneous nodes can be solved in polynomial time. They provide a dynamic programming algorithm, which even allows to add communication costs in the objective function.

2.1.2 With Homogeneous Nodes and the *Multiple* Strategy

Theorem 2.1. *The instance of the* REPLICA COUNTING *problem with the Multiple strategy can be solved in polynomial time.*

Proof. We outline below an optimal algorithm to solve the problem. The proof of optimality is quite technical, so the reader may want to skip it at first reading. ■

Algorithm for Multiple Servers

We propose a greedy algorithm to solve the REPLICA COUNTING problem. Let W be the total number of requests that a server can handle.

This algorithm works in three passes: first we select the nodes which will have a replica handling exactly W requests. Then a second pass allows us to select some extra servers which are fulfilling the remaining requests. Finally, we need to decide for each server how many requests of each client it is processing.

We assume that each node i knows its parent $\mathsf{parent}(i)$ and its children $\mathsf{children}(i)$ in the tree. We introduce a new variable which is the flow coming up in the tree (requests which are

not already fulfilled by a server). It is denoted by flow_i for the flow between i and $\mathsf{parent}(i)$. Initially, $\forall i \in \mathcal{C}$ $\mathsf{flow}_i = r_i$ and $\forall i \in \mathcal{N}$ $\mathsf{flow}_i = -1$. Moreover, the set of replicas is empty in the beginning: $repl = \emptyset$.

Pass 1– We greedily select in this step some nodes which will process W requests and which are as close to the leaves as possible. We place a replica on such nodes (see Algorithm 1). Procedure **pass1** is called with r (root of the tree) as a parameter, and it goes down the tree recursively in order to compute the flows. When a flow exceeds W, we place a replica since the corresponding server will be fully used, and we remove the processed requests from the flow going upwards.

At the end, if $flow_r = 0$ or ($flow_r \leq \mathsf{W}$ and $r \notin repl$), we have an optimal solution since all replicas which have been placed are fully used and all requests are satisfied by adding a replica in r if $flow_r \neq 0$. In this case we skip pass 2 and go directly to pass 3.

Otherwise, we need some extra replicas since some requests are not satisfied yet, and the root cannot satisfy all the remaining requests. To place these extra replicas, we go through pass 2.

Algorithm 1: Procedure pass1

```
procedure pass1 (node s ∈ 𝒩)
begin
    flow_s = 0;
    for i ∈ children(s) do
        if flow_i == −1 then pass1(i); // Recursive call.
        flow_s = flow_s + flow_i;
    end
    if flow_s ≥ W then flow_s = flow_s − W; repl = {s} ∪ repl;
end
```

Pass 2– In this pass, we need to select the nodes where to add replicas. To do so, while there are too many requests going up to the root, we select the node which can process the highest number of requests, and we place a replica there. The number of requests that a node $j \in \mathcal{N}$ can eventually process is the minimum of the flows between j and the root r, denoted $uflow_j$ (for *u*seful *flow*). Indeed, some requests may have no server yet, but they might be processed by a server on the path between j and r, where a replica has been placed in pass 1. Algorithm 2 details this pass.

If we exit this pass with $finish = -1$, this means that we have tried to place replicas on all nodes, but this solution is not feasible since there are still some requests which are not processed going up to the root. In this case, the original problem instance had no solution.

However, if we succeed to place replicas such that $flow_r = 0$, we have a set of replicas which succeed to process all requests. We then go through pass 3 to assign requests to servers, i.e., to compute how many requests of each client should be processed by each server.

Pass 3– This pass is in fact straightforward, starting from the leaves and distributing the requests to the servers from the bottom until the top of the tree. We decide for instance to affect requests from clients starting to the left. Procedure **pass3** is called with r (root

Algorithm 2: Pass 2

```
while flow_r ≠ 0 do
    freenode = N \ repl;
    if freenode == ∅ then finish = −1; exit the loop;
    // At each step, assign 1 replica and re-compute flows.
    child = children(r); uflow_r = flow_r;
    while child! = ∅ do
        remove j from child;
        uflow_j = min(flow_j, uflow_parent(j));
        child = child ∪ children(j);
    end
    // The useful flows have been computed, select the max.
    maxuflow=0;
    for j ∈ freenode do
        if uflow_j > maxuflow then maxuflow = uflow_j; maxnode = j;
    end
    if maxuflow ≠ 0 then
        repl = repl ∪ {maxnode};
        // Update the flows upwards.
        for j ∈ Ancestors(maxnode) ∪ {maxnode} do flow_j = flow_j − maxuflow;
    end
    else finish = −1; exit the loop;
end
```

of the tree) as a parameter, and it goes down the tree recursively (Cf. Algorithm 3). For $i \in \mathcal{C}$, r'_i is the number of requests of i not yet affected to a server (initially $r'_i = r_i$). $w_{s,i}$ is the number of requests of client i affected to server $s \in \mathcal{N}$, and $w_s \leq \mathsf{W}$ is the total number of requests affected to s. $C(s)$ is the set of clients in $\mathsf{subtree}(s)$ which still have some requests not affected. Initially, $C(i) = \{i\}$ for $i \in \mathcal{C}$, and $C(s) = \emptyset$ otherwise.

Note that a server which was computing W requests in pass 1 may end up computing fewer requests if one of its descendants in the tree has earned a replica in pass 2. But this does not affect the optimality of the result, since we keep the same number of replicas.

The proof in Section 2.1.2 shows the equivalence between the solution built by this algorithm and any optimal solution, thus proving the optimality of the algorithm. The following example illustrates the step by step execution of the algorithm.

Example

Figure 2.1(a) provides an example of network on which we are placing replicas with the *Multiple* strategy. The network is thus homogeneous and we fix $\mathsf{W} = 10$.

Pass 1 of the algorithm is quite straightforward to unroll, and Figure 2.1(b) indicates the flow on each link and the saturated replicas are the black nodes.

During pass 2, we select the nodes of maximum useful flow. Figure 2.1(c) represents these useful flows; we see that node n_4 is the one with the maximum useful flow (7), so we assign it a replica and update the useful flows. All the useful flows are then reduced down to 1 since there is only 1 request going through the root n_1. The first node of maximum useful flow 1 to be

Algorithm 3: Procedure pass3

```
procedure pass3 (node s ∈ N)
begin
    w_s = 0;
    for i ∈ children(s) do
        if C(i) = ∅ then pass3(i); // Recursive call.
        C(s) = C(s) ∪ C(i);
    end
    if s ∈ repl then
        for i ∈ C(s) do
            if r'(i) ≤ W − w_s then C(s) = C(s) \ {i}; w_{s,i} = r'_i; w_s = w_s + r'_i; r'_i = 0;
        end
        if C(s) ≠ ∅ then Let i ∈ C(s); x = W − w_s; r'_i = r'_i − x; w_{s,i} = x; w_s = W;
    end
end
```

selected is n_2, which is set to be a replica of pass 2. The flow at the root is then 0 and it is the end of pass 2.

Finally, pass 3 affects the servers to the clients and decides which requests are served by which replica (Figure 2.1(d)). For instance, the client with 12 requests shares its requests between n_{10} (10 requests) and n_2 (2 requests). Requests are affected from the bottom of the tree up to the top. Note that the root n_1, even though it was a saturated replica of pass 1, has only 5 requests to proceed in the end.

Proof of Optimality

Let R_{opt} be an optimal solution to an instance of the problem. The core of the proof consists in transforming this solution into an equivalent canonical optimal solution R_{can}. We will then show that our algorithm is building this canonical solution, and thus it is producing an optimal solution.

Each server $s \in R_{opt}$ is serving $w_{s,i}$ requests of client $i \in \mathsf{subtree}(s) \cap \mathcal{C}$, and

$$w_s = \sum_{i \in \mathsf{subtree}(s) \cap \mathcal{C}} w_{s,i} \leq W.$$

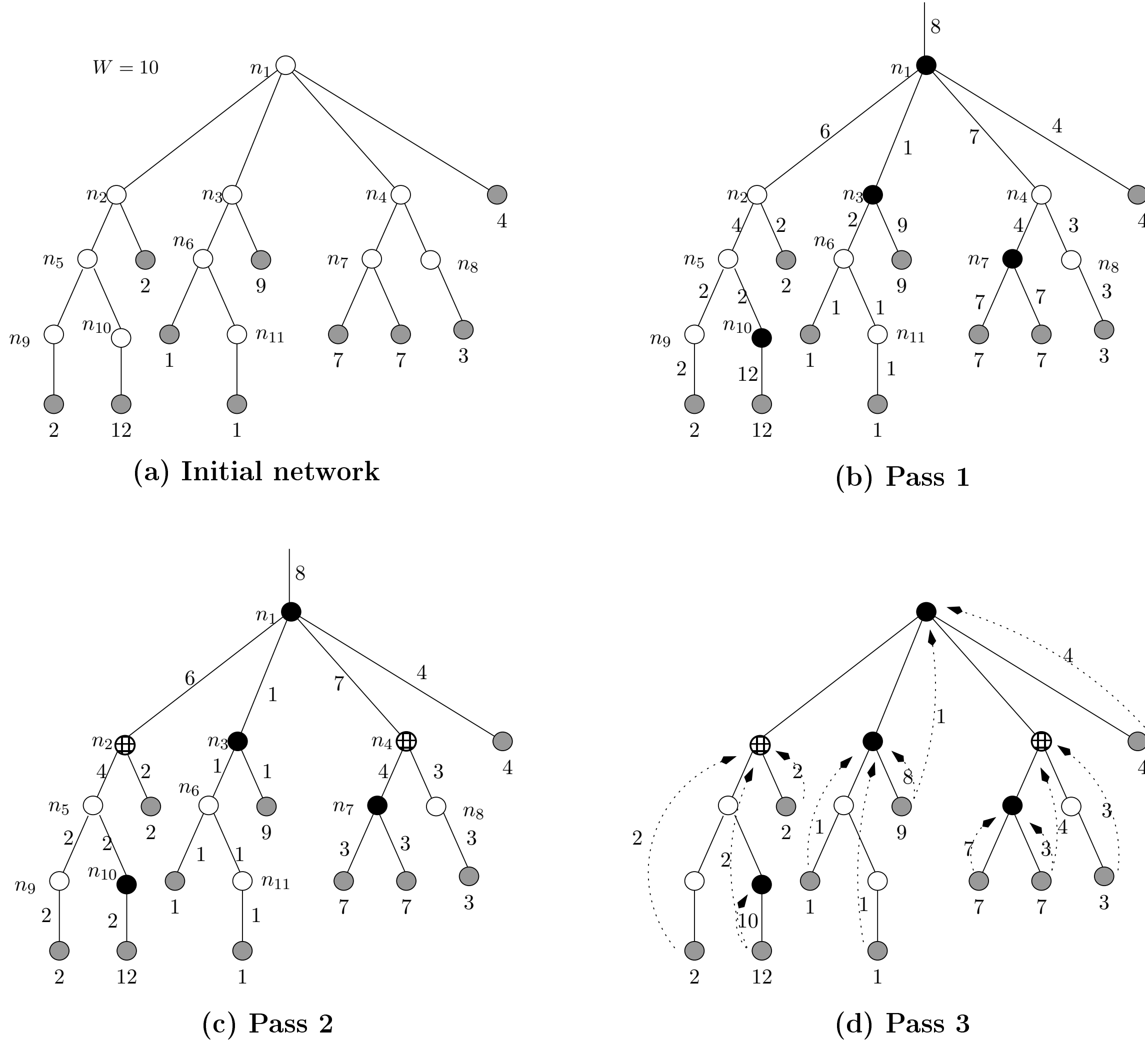
For each $i \in \mathcal{C}$, $w_{s,i} = 0$ if $s \in \mathcal{N}$ is not a replica, and, $\sum_{s \in Ancests(i)} w_{s,i} = r_i$.

We define the *flow* of node k, flow_k, by the number of requests going through this node up to its parents. Thus, for $i \in \mathcal{C}$, $flow_i = r_i$, while for a node $s \in \mathcal{N}$,

$$flow_s = \sum_{i \in \mathsf{children}(s)} flow_i - w_s.$$

The *total flow* going through the tree, $tflow$, is defined in a similar way, except that we do not remove from the flow the requests processed by a replica, i.e., $tflow_s = \sum_{i \in \mathsf{children}(s)} tflow_i$. We thus have

$$tflow_s = \sum_{i \in \mathsf{subtree}(s) \cap \mathcal{C}} r_i.$$



(a) Initial network
(b) Pass 1
(c) Pass 2
(d) Pass 3

Figure 2.1: Algorithm for the Replica Counting problem with the *Multiple* strategy.

These variables are completely defined by the network and the optimal solution R_{opt}.

A first lemma shows that it is possible to change request assignments while keeping an optimal solution. The flows need to be recomputed after any such modification.

Lemma 2.1. *Let $s \in \mathcal{N} \cap R_{opt}$ be a server such that $w_s < \mathsf{W}$.*

- *If $tflow_s \geq \mathsf{W}$, we can change the request assignment between replicas of the optimal solution, in such a way that $w_s = \mathsf{W}$.*
- *Otherwise, we can change the request assignment so that $w_s = tflow_s$.*

Proof. First we point out that the clients in $\mathsf{subtree}(s)$ can all be served by s, and since R_{opt} is a solution, these requests are served by a replica somewhere in the tree. We do not modify the optimality of the solution by changing the $w_{s,i}$, it just affects the flows of the solution. Thus, for a given client $i \in \mathsf{subtree}(s) \cap \mathcal{C}$, if there is a replica $s' \neq s$ on the path between i and the root, we can change the assignment of the requests of client i. Let $x = max(w_{s',i}, \mathsf{W} - w_s)$. Then we move x requests, i.e., $w_{s',i} = w_{s',i} - x$ and $w_{s,i} = w_{s,i} + x$. From the definition of $tflow_s$, we

obtain the result, if we move all possible requests to s until there are no more requests in the subtree or until s is processing W requests. ■

We now introduce a new definition, completely independent from the optimal solution but related to the tree network. The *canonical flow* is obtained by distinguishing nodes which receive a flow greater than W from the other nodes. We compute the canonical flow $cflow$ of the tree, independently of the replica placement, and define a subset of nodes which are *saturated*, SN. We also compute the number of saturated nodes in $\mathsf{subtree}(k)$, denoted nsn_k, for any node $k \in \mathcal{C} \cup \mathcal{N}$ of the tree.

For $i \in \mathcal{C}$, $cflow_i = r_i$ and $nsn_i = 0$, and we then compute recursively the canonical flows for nodes $s \in \mathcal{N}$. Let $f_s = \sum_{i \in \mathsf{children}(s)} cflow_i$ and $x_s = \sum_{i \in \mathsf{children}(s)} nsn_i$. If $f_s \geq \mathsf{W}$ then $s \in SN$, $cflow_s = f_s - \mathsf{W}$ and $nsn_s = x_s + 1$. Otherwise, s is not saturated, $cflow_s = f_s$ and $nsn_s = x_s$.

We can deduce from these definitions the following results:

Proposition 2.1. *A non saturated node always has a canonical flow being less than W:*
$\forall s \in \mathcal{N} \setminus SN \;\; cflow_s < \mathsf{W}$

Lemma 2.2. *For all nodes $s \in \mathcal{C} \cup \mathcal{N}$, $cflow_s = tflow_s - nsn_s \times \mathsf{W}$.*

Corollary 2.1. *For all nodes $s \in \mathcal{C} \cup \mathcal{N}$, $tflow_s \geq nsn_s \times \mathsf{W}$.*

Proof. Proposition 2.1 is trivial due to the definition of the canonical flow.

Lemma 2.2 can be proved recursively on the tree.

- This property is true for the clients: for $i \in \mathcal{C}$, $nsn_i = 0$ and $tflow_i = cflow_i = r_i$.
- Let $s \in \mathcal{N}$, and let us assume that the proposition is true for all children of s. Then,

$$\forall j \in \mathsf{children}(s) \; cflow_j = tflow_j - nsn_j \times \mathsf{W}.$$

  - If $s \notin SN$, $nsn_s = \sum_{j \in \mathsf{children}(s)} nsn_j$ and

$$cflow_s = \sum_{j \in \mathsf{children}(s)} cflow_j = \sum_{j \in \mathsf{children}(s)} (tflow_j - nsn_j \times \mathsf{W}) = tflow_s - nsn_s \times \mathsf{W}$$

  - If $s \in SN$, $nsn_s = \left(\sum_{j \in \mathsf{children}(s)} nsn_j\right) + 1$ and

$$cflow_s = \sum_{j \in \mathsf{children}(s)} cflow_j - \mathsf{W} = \sum_{j \in \mathsf{children}(s)} (tflow_j - nsn_j \times \mathsf{W}) - \mathsf{W}$$

$$= tflow_s - (nsn_s - 1) \times \mathsf{W} - \mathsf{W} = tflow_s - nsn_s \times \mathsf{W}$$

which proves the result. Corollary 2.1 is trivially deduced from Lemma 2.2 since $cflow$ is a positive function. ■

We also show that it is always possible to move a replica into a free server which is one of its ancestors in the tree, while keeping an optimal solution:

Proposition 2.2. *Let R_{opt} be an optimal solution, and let $s \in R_{opt}$. If $\exists s' \in \mathsf{Ancestors}(s) \setminus R_{opt}$ then $R'_{opt} = \{s'\} \cup R_{opt} \setminus \{s\}$ is also an optimal solution.*

Proof. s' can handle all requests which were processed by s since $s \in \mathsf{subtree}(s')$. We just need to redefine $w_{s',i} = w_{s,i}$ for all $i \in \mathcal{C}$ and then $w_{s,i} = 0$. ■

We are now ready to transform R_{opt} into a new optimal solution, R_{sat}, by redistributing the requests among the replicas and moving some replicas, in order to place a replica at each saturated node, and affecting W requests to this replica. This transformation is done starting at the leaves of the tree, and considering all nodes of SN. Nothing needs to be done for the leaves (the clients) since they are not in SN.

Let us consider $s \in SN$, and assume that the optimal solution has already been modified to place a replica, and assign it W requests, on all nodes in $subSN = SN \cap \mathsf{subtree}(s) \setminus \{s\}$.

We need to differentiate two cases:

1. If $s \in R_{opt}$, we do not need to move any replica. However, if $w_s \neq \mathsf{W}$, we change the assignment of some requests while keeping the same replicas in order to obtain a workload of W on server s. We do not remove requests from the saturated servers of $subSN$ which have already been filled. Corollary 2.1 ensures that $tflow_s \geq nsn_s \times \mathsf{W}$, and $(nsn_s - 1) \times \mathsf{W}$ requests should not move since they are affected to the $nsn_s - 1$ servers of $subSN$. There are thus still more than W requests of clients of $\mathsf{subtree}(s)$ which can possibly be moved on s using Lemma 2.1.
2. If $s \notin R_{opt}$, we need to move a replica of R_{opt} and place it in s without changing the optimality of the solution. We differentiate two subcases.
   (a) If $\exists s_1 \in \mathsf{subtree}(s) \cap R_{opt} \setminus SN$, then the replica placed on s_1 can be moved in s by applying Proposition 2.2. Then, if $w_s \neq \mathsf{W}$, we apply case 1 above to saturate the server.
   (b) Otherwise, all the replicas placed in $\mathsf{subtree}(s)$ are also in SN, and the flow consumed by the already modified optimal algorithm is exactly $(nsn_s - 1) \times \mathsf{W}$. It is easy to see that the flow (of the optimal solution) at s is exactly equal to the total flow minus the consumed flow. Therefore, $flow_s = tflow_s - (nsn_s - 1) \times \mathsf{W}$, and with the application of Corollary 2.1, $flow_s \geq \mathsf{W}$.
   The idea now consists in affecting the requests of this flow to node s by removing work from the replicas upwards to the root, and rearrange the remaining requests to remove one replica. The flow $flow_s$ is going upwards to be processed by some of the nr_s replicas in $\mathsf{Ancestors}(s) \cap R_{opt}$, denoted $s_1, ..., s_{nr_s}$, s_1 being the closest node from s. We can remove W of these requests from the flow and affect them to a new replica placed in s. Let $w_{s_k,s} = \sum_{j \in \mathsf{subtree}(s) \cap \mathcal{C}} w_{s_k,j}$. We have $\sum_{k=1..nr_s} w_{s_k,s} = flow_s$. We move these requests from s_k to s, starting with $k = 1$. Thus, after the modification, $w_{s_1,s} = 0$. It is however possible that $w_{s_1} \neq 0$ since s_1 may process requests which are not coming from $\mathsf{subtree}(s)$. In this case, we are sure that we have removed enough requests from s_k, $k = 2..nr_s$ which can instead process requests still in charge of s_1. We can then remove the replica initially placed in s_1.
   This way, we have not changed the assignment on replicas in $subSN$, but we have placed a replica in s which is processing W requests. Since we have at the same time removed the first replica on the path from s to the root (s_1), we have not changed the number of replicas and the solution is still optimal.

Once we have applied this procedure up to the root, we have an optimal solution R_{sat} in which all nodes of SN have been placed a replica and are processing W requests. We will not change the assignment of these replicas anymore in the following. Free nodes in the new solution are called *F-nodes*, while replicas which are not in SN are called *PS-nodes*, for *partially saturated*.

In a next step, we further modify the R_{sat} optimal solution in order to obtain what we call

the *canonical solution* R_{can}. To do so, we change the request assignment of the PS-nodes: we "saturate" some of them as much as we can and we integrate them into the subset of nodes SN, redefining the $cflow$ accordingly. At the end of the process, $SN = R_{can}$.

The $cflow$ is still the flow which has not been processed by a saturated node in the subtree, and thus we can express it in a more general way:

$$cflow_s = tflow_s - \sum_{s' \in SN \cap \mathsf{subtree}(s)} w_{s'}$$

Note that this is totally equivalent to the previous definition while we have not modified SN.

We also introduce a new flow definition, the *non-saturated flow* of s, $nsflow_s$, which counts the requests going through node s and not served by a saturated server anywhere in the tree. Thus,

$$nsflow_s = cflow_s - \sum_{i \in \mathsf{children}(s) \cap \mathcal{C}} \sum_{s' \in \mathsf{Ancestors}(s) \cap SN} w_{s',i}.$$

This flow represents the requests that can potentially be served by s while keeping all nodes of SN saturated.

Lemma 2.3. *In a saturated optimal solution, there cannot exist a PS-node in the subtree of another PS-node.*

Proof. The non-saturated flow is $nsflow_s \leq cflow_s$ since we further remove from the canonical flow some requests which are affected upwards in the tree to some saturated servers.

Let $s \in R_{sat} \setminus SN$ be a PS-node. Its canonical flow is $cflow_s < W$. It can potentially process all the requests of the subtree which are not affected to a saturated server upwards or downwards in the tree, thus $nsflow_s$ requests. Since $nsflow_s \leq cflow_s < W$, we can change the request assignment to assign all these $nsflow_s$ requests to s, removing eventually some work from other non-saturated replicas upwards or downwards which were processing these requests. Thus, the replica on node s is processing all the requests of $\mathsf{subtree}(s)$ which are not processed by saturated nodes.

If there was a non saturated replica in $\mathsf{subtree}(s)$, it could thus be removed since all the requests are processed by s. This means that a solution with a PS-node in the subtree of another PS-node is not optimal, thus proving the lemma. ■

At this point, we can move the PS-nodes as high as possible in R_{sat}. Let s be a PS-node. If there is a free node s' in $\mathsf{Ancestors}(s)$ then we can move the replica from s to s' using Proposition 2.2. Lemma 2.3 ensures that there are no other PS-nodes in $\mathsf{subtree}(s')$.

All further modifications will only alter nodes which have no PS-nodes in their ancestors. We define $\mathcal{N}' = \{s | \mathsf{Ancestors}(s) \setminus SN = \emptyset\}$.

Let $s \in \mathcal{N}'$. $nsflow_s = cflow_s - \sum_{i \in \mathsf{children}(s) \cap \mathcal{C}} \sum_{s' \in \mathsf{Ancestors}(s)} w_{s',i}$ since all ancestors of s are in SN. Thus,

$$nsflow_s = \sum_{s' \in \mathsf{subtree}(s) \setminus SN} w_{s'}.$$

By definition, $\forall s \in \mathcal{N}$ $nsflow_s \leq cflow_s$. Moreover, if $s \notin SN$, then $nsflow_s = w_s$ since $\mathsf{subtree}(s) \setminus SN$ is reduced to s (no other PS-node under the PS-node s, from Lemma 2.3).

We introduce a new flow definition, the useful flow, which intuitively represents the number of requests that can possibly be processed on s without removing requests from a saturated server.

$$uflow_s = \min_{s' \in \mathsf{Ancestors}(s) \cup \{s\}} \{cflow_{s'}\}$$

Lemma 2.4. *Let $s \in \mathcal{N}'$. Then $nsflow_s \leq uflow_s$.*

Proof. Let $s' \in \mathsf{Ancestors}(s)$. Since $s \in \mathcal{N}'$, $s' \in SN$.

$$cflow_{s'} \geq nsflow_{s'} = \sum_{s'' \in \mathsf{subtree}(s') \setminus SN} w_{s''}$$

But since $s \in \mathsf{subtree}(s')$, $\mathsf{subtree}(s) \setminus SN \subseteq \mathsf{subtree}(s') \setminus SN$, hence $nsflow_s \leq nsflow_{s'}$. Note that $nsflow$ is a non decreasing function (when going up the tree).

Thus, $\forall s' \in \mathsf{Ancestors}(s) \cup \{s\}$, $nsflow_s \leq cflow_{s'}$, and by definition of the useful flow, $nsflow_s \leq uflow_s$. ■

Now we start the modification of the optimal solution in order to obtain the canonical solution. At each step, we select a node $s \in \mathcal{N} \setminus SN$ maximizing the useful flow. If there are several nodes of identical $uflow$, we select the first one in a depth-first traversal of the tree. We will prove that we can affect $uflow_s$ requests to this node without unsaturating any server of SN. s is then considered as a saturated node, we recompute the canonical flows (and thus the useful flows) and reiterate the process until $cflow_r = 0$, which means that all the requests have been affected to saturated servers.

Let us explain how to reassign the requests in order to saturate s with $uflow_s$ requests. The idea is to remove some requests from $\mathsf{Ancestors}(s)$ in order to saturate s, and then to saturate the ancestors of s again, by affecting them some requests coming from other non saturated servers.

First, we note that $uflow_s \leq cflow_r = nsflow_r$. Thus,

$$uflow_s \leq \sum_{s' \in \mathcal{N} \setminus SN} w_{s'} = w_s + \sum_{s' \in PS} w_{s'}$$

where PS is the set of non saturated nodes without s. Let $x = uflow_s - w_s$. If $x = 0$, s is already saturated. Otherwise, we need to reassign x requests to s. From the previous equation, we can see that $\sum_{s' \in PS} w_{s'} \geq uflow_s - w_s = x$. There are thus enough requests handled by non saturated nodes which can be passed to s.

The number of requests of $\mathsf{subtree}(s) \cap \mathcal{C}$ handled by $\mathsf{Ancestors}(s)$ is

$$\sum_{s' \in \mathsf{Ancestors}(s)} \sum_{i \in \mathsf{subtree}(s) \cap \mathcal{C}} w_{s',i} = cflow_s - nsflow_s$$

by definition of the flow. Or $cflow_s - nsflow_s \geq uflow_s - w_s = x$ so there are at least x requests that s can take from its ancestors.

Let $a_1 = \mathsf{parent}(s), ..., a_k = r$ be the ancestors of s. $x_j = \sum_{i \in \mathsf{subtree}(s) \cap \mathcal{C}} w_{a_j,i}$ is the amount of requests that s can take from a_j. We choose arbitrary where to take the requests if $\sum_j x_j > x$, and do not modify the assignment of the other requests. We thus assume in the following that $\sum_j x_j = x$. Since these x_j requests are coming from a client in $\mathsf{subtree}(s)$, we can assign them

to s, and there are now only $\mathsf{W}-x_j$ requests handled by a_j, which means that a_j is temporarily unsaturated. However, we have given x extra requests to s, hence s is processing $w_s+x = uflow_s$ requests.

We finally need to reassign requests to $a_j, j = 1..k$ in order to saturate these nodes again, taking requests out of nodes in PS (non saturated nodes other than s). This is done iteratively starting with $j = 1$ and going up to the root a_k. At each step j, we assume that $a_{j'}, j' < j$ have already been saturated again and we should not move requests away from them. However, we can still eventually take requests away from $a_{j''}, j'' > j$.

In order to saturate a_j, we need to take:

- either requests from $\mathsf{subtree}(a_j) \cap \mathcal{C}$ which are currently handled by $a_{j''}, j'' > j$, but without moving requests which are already affected to s (i.e., $\sum_{j''>j} x_{j''}$);
- or requests from non saturated servers in $\mathsf{subtree}(a_j)$, except requests from s and requests already given to s that should not be moved any more (i.e., $\sum_{j'<j} x_{j'}$).

The number of requests that we can potentially affect to a_j is therefore:

$$X = \sum_{s' \in \mathsf{subtree}(a_j) \setminus SN \setminus \{s\}} w_{s'} + \sum_{i \in \mathsf{subtree}(a_j) \cap \mathcal{C}} \sum_{s' \in \mathsf{Ancestors}(a_j)} w_{s',i} - \sum_{j'<j} x_{j'} - \sum_{j''>j} x_{j''}$$

Let us show that $X \geq x_j$. Then we can use these requests to saturate a_j again.

$$cflow_{a_j} = nsflow_{a_j} + \sum_{i \in \mathsf{subtree}(a_j) \cap \mathcal{C}} \sum_{s' \in \mathsf{Ancestors}(a_j)} w_{s',i} = w_s + X + \sum_{j'<j} x_{j'} + \sum_{j''>j} x_{j''} = X + w_s + x - x_j$$

But $cflow_{a_j} \geq uflow_s$ and $uflow_s - w_s = x$ so

$$X = cflow_{a_j} - w_s - x + x_j \geq uflow_s - w_s - x + x_j = x_j$$

It is thus possible to saturate s and then keep its ancestors saturated. At this point, s becomes a node of SN and we can recompute the canonical and non saturated flows. We have removed $uflow_s$ requests which were processed by non saturated servers, so the $cflow$ and $nsflow$ of all ancestors of s, including s, should be decreased by $uflow_s$.

In particular, at the root, $cflow_r = cflow_r - uflow_s$, which proves that the contribution of s on $cflow_r$ is $uflow_s$.

In the last step of the proof, we show that the number of replicas in the modified canonical solution at the end of the iteration $R_{can} = SN$ has exactly the same number of replicas than R_{sat}. In the saturated solution, each PS-node s is processing $nsflow_s$ requests, while in the canonical solution, it is $uflow_s$. However, at every step when adding a saturated node s, we have $uflow_s$ greater than any of the $nsflow$s. It is thus easy to see that the number of nodes in the canonical solution is less or equal to the number of nodes in the saturated solution. Since the saturated solution is optimal, $|R_{can}| = |R_{sat}|$, which completes the proof.

Our algorithm builds R_{can} in polynomial time, which assesses the complexity of the problem.

2.1.3 With Homogeneous Nodes and the *Upwards* Strategy

Theorem 2.2. *The instance of the* Replica Counting *problem with the Upwards strategy is NP-complete in the strong sense.*

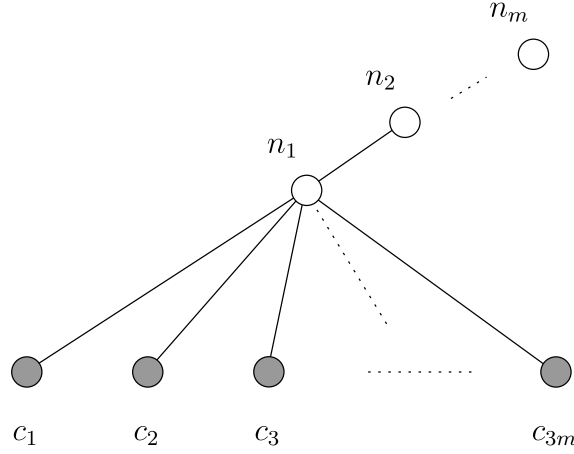


Figure 2.2: The platform used in the reduction for Theorem 2.2.

Proof. The problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that all requests are served and that no server capacity is exceeded. To establish the completeness in the strong sense, we use a reduction from 3-PARTITION [32]. We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given $3m$ positive integers $a_1, a_2, \ldots, a_{3m}$ such that $B/4 < a_i < B/2$ for $1 \leq i \leq 3m$, and $\sum_{i=1}^{3m} a_i = mB$, can we partition these integers into m triples, each of sum B? We build the following instance $\mathcal{I}_2$ of Replica Counting (see Figure 2.2):

- $3m$ clients c_i with $r_i = a_i$ for $1 \leq i \leq 3m$.
- m internal nodes n_j with $\mathsf{W}_j = \mathsf{sc}_j = B$ for $1 \leq j \leq m$.
  - The children of n_1 are all the $3m$ clients c_i, and its parent is n_2.
  - For $2 \leq j \leq m$, the only child of n_j is n_{j-1}. For $1 \leq j \leq m-1$, the parent of n_j is n_{j+1} (hence n_m is the root).

Finally, we ask whether there exists a solution with total storage cost mB, i.e., with a replica located at each internal node. Clearly, the size of $\mathcal{I}_2$ is polynomial (and even linear) in the size of $\mathcal{I}_1$.

We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution. Let $(a_{k_1}, a_{k_2}, a_{k_3})$ be the k-triplet in $\mathcal{I}_1$. We assign the three clients c_{k_1}, c_{k_2} and c_{k_3} to server n_k. Because $a_{k_1} + a_{k_2} + a_{k_3} = B$, no server capacity is exceeded. Because the m triples partition the a_i, all requests are satisfied. We do have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Let I_k be the set of clients served by node n_k if there is a replica located at n_k: then $\sum_{i \in I_k} a_i \leq B$. The total number of requests to be satisfied is $\sum_{i=1}^{3m} a_i = mB$, and there are at most m replicas of capacity B. Hence no set I_k can be empty, and $\sum_{i \in I_k} a_i \leq B$ for $1 \leq k \leq m$. Because $B/4 < a_i < B/2$, each I_k must be a triple. This leads to the desired solution of $\mathcal{I}_1$. ∎

2.1.4 With Heterogeneous Nodes

Theorem 2.3. *All three instances of the* Replica Cost *problem with heterogeneous nodes are NP-complete.*

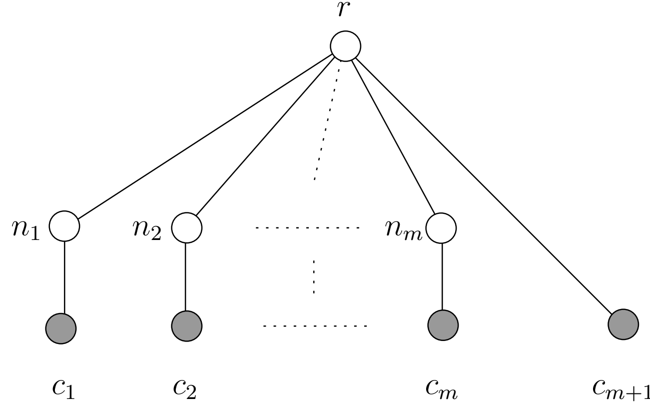


Figure 2.3: The platform used in the reduction for Theorem 2.3.

Proof. Obviously, the NP-completeness of the *Upwards* strategy is a consequence of Theorem 2.2. For the other two strategies, the problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that all requests are served and that no server capacity is exceeded. To establish the completeness, we use a reduction from 2-PARTITION [32]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION: given m positive integers $a_1, a_2, \ldots, a_m$, does there exist a subset $I \subset \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{m} a_i$. We build the following instance $\mathcal{I}_2$ of Replica Cost (see Figure 2.3):

- $m+1$ clients c_i with $r_i = a_i$ for $1 \le i \le m$ and $r_{m+1} = 1$.
- $m+1$ internal nodes:
  - m nodes n_j, $1 \le j \le m$, with $\mathsf{W}_j = \mathsf{sc}_j = a_j$.
  - A root node r with $\mathsf{W}_r = \mathsf{sc}_r = S/2 + 1$. - The only child of n_j is c_j. The parent of n_j is r. The parent of c_{n+1} is r.

Finally, we ask whether there exists a solution with total storage cost $S+1$. Clearly, the size of $\mathcal{I}_2$ is polynomial (and even linear) in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does. The same reduction works for both strategies, *Closest* and *Multiple*.

Suppose first that $\mathcal{I}_1$ has a solution. We assign a replica to each node n_i, $i \in I$, and one in the root r. Client c_i is served by n_i if $i \in \mathcal{I}$, and by the root r otherwise, i.e., if $i \notin \mathcal{I}$ or if $i = m+1$. The total storage cost is $\sum_{j \in I} W_j + W_r = S + 1$. Because $\mathsf{W}_r = S/2 + 1 = \sum_{i \notin I} r_i + r_{n+1}$, the capacity of the root is not exceeded. Note that the server allocation is compatible both with the *Closest* and *Multiple* policies. In both cases, we have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Necessarily, there is a replica located in the root, otherwise client c_{n+1} would not be served. Let I be the index set of nodes n_j, $1 \le j \le n$, which have been allocated a replica in the solution of $\mathcal{I}_2$. For $j \notin I$, there is no replica in node n_j, hence all requests of client c_j are processed by the root, whose storage capacity is $S/2+1$. We derive that $\sum_{j \notin I} r_j \le S/2$. Because the total storage capacity is $S+1$, the total storage capacity of nodes in I is $S/2$. The proof is slightly different for the two server strategies:

- For the *Closest* strategy, all requests from a client $c_j \in I$ are served by n_j, hence $\sum_{j \in I} r_j \le S/2$. Since $\sum_{j \in I} r_j + \sum_{j \notin I} r_j = S$, we derive $\sum_{j \in I} r_j = \sum_{j \notin I} r_j = S/2$, hence a solution to $\mathcal{I}_2$.

- For the *Multiple* strategy, consider a server $j \in I$. Let r'_j be the number of requests from client c_j served by n_j, and r''_j be the number of requests from c_j served by the root r (of course $r_j = r'_j + r''_j$). All requests from a client c_j, $j \notin I$, are served by the root. Let $A = \sum_{j \in I} r'_j$, $B = \sum_{j \in I} r''_j$ and $C = \sum_{j \notin I} r_j$. The total storage cost is $A + B + S/2 + 1$, hence $A + B \leq S/2$. We have seen that $C \leq S/2$. But $A + B + C = S$, hence $B = 0$, and $A = C = S/2$, hence a solution to $\mathcal{I}_2$.

■

2.2 Linear Programming Formulation

In this section, we express the Replica Placement optimization problem in terms of an integer linear program (LP). We derive a formulation for each of the three server access policies, namely *Closest*, *Upwards* and *Multiple*. This is an important extension to a previous formulation due to Karlsson et al. [42] as they restrict to the *Closest* access policy.

While there is no efficient algorithm to solve integer linear programs (unless P=NP), this formulation is extremely useful as it leads to an absolute lower bound: we solve the integer linear program over the rationals, using standard software packages [17, 34]. Of course the rational solution will not be feasible, as it assigns fractions of replicas to server nodes, but it will provide a lower bound on the storage cost of any solution. This bound will be very helpful to assess the performance of the polynomial heuristics that are introduced in Section 2.3.

2.2.1 Single Server

We start with single server strategies, namely the *Upwards* and *Closest* access policies. We need to define a few variables:

Server assignment

- x_j is a boolean variable equal to 1 if j is a server (for one or several clients).
- $y_{i,j}$ is a boolean variable equal to 1 if $j = \mathsf{server}(i)$.
- If $j \notin \mathsf{Ancestors}(i)$, we directly set $y_{i,j} = 0$.

Link assignment

- $z_{i,l}$ is a boolean variable equal to 1 if link $l \in \mathsf{path}[i \to r]$ is used when client i accesses its server $\mathsf{server}(i)$.
- If $l \notin \mathsf{path}[i \to r]$ we directly set $z_{i,l} = 0$.

The objective function is the total storage cost, namely $\sum_{j \in \mathcal{N}} \mathsf{sc}_j x_j$. We list below the constraints common to the *Closest* and *Upwards* policies: First there are constraints for server and link usage:

- Every client is assigned a server:

$$\forall i \in \mathcal{C}, \sum_{j \in \mathsf{Ancestors}(i)} y_{i,j} = 1$$

- All requests from $i \in \mathcal{C}$ use the link to its parent:

$$z_{i,i \to \mathsf{parent}(i)} = 1$$

- Let $i \in \mathcal{C}$, and consider any link $l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r]$. If $j' = \mathsf{server}(i)$ then link $\mathsf{succ}(l)$ is not used by i (if it exists). Otherwise $z_{i,\mathsf{succ}(l)} = z_{i,l}$. Thus:

$$\forall i \in \mathcal{C}, \forall l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r],$$

$$z_{i,\mathsf{succ}(l)} = z_{i,l} - y_{i,j'}$$

Next there are constraints expressing that server capacities cannot be exceeded: $\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} r_i y_{i,j} \leq \mathsf{W}_j x_j$. Note that this ensures that if j is the server of i, there is indeed a replica located in node j. Altogether, we have fully characterized the linear program for the *Upwards* policy. We need additional constraints for the *Closest* policy, which is a particular case of the *Upwards* policy (hence all constraints and equations remain valid).

We need to express that if node j is the server of client i, then no ancestor of j can be the server of a client in the subtree rooted at j. Indeed, a client in this subtree would need to be served by j and not by one of its ancestors, according to the *Closest* policy. A direct way to write this constraint is

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i),$$

$$\forall i' \in \mathcal{C} \cap \mathsf{subtree}(j), \forall j' \in \mathsf{Ancestors}(j),$$

$$y_{i,j} \leq 1 - y_{i',j'}$$

Indeed, if $y_{i,j} = 1$, meaning that $j = \mathsf{server}(i)$, then any client i' in the subtree rooted in j must have its server in that subtree, not closer to the root than j. Hence $y_{i',j'} = 0$ for any ancestor j' of j.

There are $O(s^4)$ such constraints to write, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size. We can reduce this number down to $O(s^3)$ by writing

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i) \setminus \{r\},$$

$$\forall i' \in \mathcal{C} \cap \mathsf{subtree}(j),$$

$$y_{i,j} \leq 1 - z_{i',j \to \mathsf{parent}(j)}$$

2.2.2 Multiple Servers

We now proceed to the *Multiple* policy. We define the following variables:

Server assignment

- x_j is a boolean variable equal to 1 if j is a server (for one or several clients).
- $y_{i,j}$ is an integer variable equal to the number of requests from client i processed by node j.
- If $j \notin \mathsf{Ancestors}(i)$, we directly set $y_{i,j} = 0$.

Link assignment

- $z_{i,l}$ is an integer variable equal to the number of requests flowing through link $l \in \mathsf{path}[i \to r]$ when client i accesses any of its servers in $\mathsf{Servers}(i)$
- If $l \notin \mathsf{path}[i \to r]$ we directly set $z_{i,l} = 0$.

The objective function is unchanged, as the total storage cost still writes $\sum_{j \in \mathcal{N}} \mathsf{sc}_j x_j$. But the constraints must be modified. First those for server and link usage:

- Every request is assigned a server:

$$\forall i \in \mathcal{C}, \sum_{j \in \mathsf{Ancestors}(i)} y_{i,j} = r_i$$

- All requests from $i \in \mathcal{C}$ use the link to its parent:

$$z_{i,i \to \mathsf{parent}(i)} = r_i$$

- Let $i \in \mathcal{C}$, and consider any link $l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r]$. Some of the requests from i which flow through l will be processed by node j', and the remaining ones will flow upwards through link $\mathsf{succ}(l)$:

$$\forall i \in \mathcal{C}, \forall l : j \to j' = \mathsf{parent}(j) \in \mathsf{path}[i \to r],$$

$$z_{i,\mathsf{succ}(l)} = z_{i,l} - y_{i,j'}$$

The other constraints on server capacities are slightly modified:

$$\forall j \in \mathcal{N}, \sum_{i \in \mathcal{C}} y_{i,j} \leq \mathsf{W}_j x_j$$

Note that this ensure that if j is the server for one or more requests from i, there is indeed a replica located in node j. Altogether, we have fully characterized the linear program for the *Multiple* policy.

2.2.3 A Mixed Integer LP-Based Lower Bound

The previous linear programs contain boolean or integer variables, because it does not make sense to assign half a request or to place one third of a replica on a node. It has to be solved in integer values if we wish to obtain an exact solution to an instance of the problem. This can be done for each access policy, but due to the large number of variables, the problem cannot be solved for platforms of size $s = |\mathcal{C}| + |\mathcal{N}| > 50$. Thus we cannot use this approach for large-scale problems.

However, we can still relax the constraints and solve the linear program assuming that all variables take rational values. The optimal solution of the relaxed program can be obtained in polynomial time (in theory using the ellipsoid method [60], in practice using standard software packages [17, 34]), and the value of its objective function provides an absolute lower bound on the cost of any valid (integer) solution. Of course the relaxation makes the most sense for the *Multiple* policy, because several fractions of servers are assigned by the rational program. For all practical values of the problem size, the rational linear program returns a solution in a few

minutes. We tested up to several thousands of nodes and clients, and we always found a solution within ten seconds.

However, we can obtain a more precise lower bound for trees with up to $s = 400$ nodes and clients by using a rational solution of the *Multiple* instance of the linear program with fewer integer variables. We treat the $y_{i,j}$ and $z_{i,l}$ as rational variables, and only require the x_j to be integer variables. These variables are set to 1 if and only if there is a replica on the corresponding node. Thus, forbidding to set $0 < x_j < 1$ allows us to get a realistic value of the cost of a solution of the problem. For instance, a server might be used only at 50% of its capacity, thus setting $x = 0.5$ would be enough to ensure that all requests are processed; but in this case, the cost of placing the replica at this node is halved, which is incorrect: while we can place a replica or not but it is impossible to place half of a replica.

In practice, this lower bound provides a drastic improvement over the unreachable lower bound provided by the fully rational linear program. The good news is that we can compute the refined lower bound for problem sizes up to $s = 400$, using GLPK [34]. We used the refined bound for all our experiments.

2.3 Heuristics for the Replica Cost Problem

In this section several heuristics for the *Closest*, *Upwards* and *Multiple* policies are presented. As previously stated, our main objective is to provide an experimental assessment of the relative performance of the three policies. All the eight heuristics described below have a worst case quadratic complexity $O(s^2)$, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size. Indeed, all heuristics proceed by traversing the tree, and the number of traversals is bounded by the number of internal nodes (and is much lower in practice).

The pseudo-code for the algorithms is provided in Appendix A.

2.3.1 *Closest*

The first two heuristics enforce the *Closest* policy through a top-down approach, whereas the third heuristic uses a bottom-up approach.

1. Closest Top Down All (CTDA) – The basic idea is to perform a breadth-first traversal of the tree. Every time a node is able to process the requests of all the clients in its subtree, the node is chosen as a server, and we do not explore further that subtree. The corresponding procedure (Algorithm 9) is called until no more servers are added in a tree traversal.

2. Closest Top Down Largest First (CTDLF) – The tree is traversed in breadth-first manner similarly to CTDA, but we treat the subtree which contains the most requests first. Also, the tree traversal is stopped each time a replica has been placed (and then the procedure is called again).

3. Closest Bottom Up (CBU) – Still dealing with the *Closest* policy, this heuristic performs a bottom-up traversal of the tree. A node is chosen as server if it can process all the requests of the clients in its subtree. Algorithm 10 describes the recursive implementation.

2.3.2 *Upwards*

We propose two heuristics for the *Upwards* policy, one using a top-down approach, the other using a bottom-up approach.

4. Upwards Top Down (UTD) – The top down approach works in two passes. In the first pass (see Algorithm 12), each node whose capacity is exhausted by the number of requests in its subtree is chosen by traversing the tree in depth-first manner. When a server is chosen, we delete as much clients as possible in non-increasing order of their r_i-values, until the server capacity is reached or no other client can be deleted. The delete-procedure is described in Algorithm 11. If not all requests can be treated by the chosen servers, a second pass is started. In this procedure (see Algorithm 13) servers with remaining requests are added.

5. Upwards Big Client First (UBCF) – The second heuristic for the *Upwards* policy works in a completely different way than all the other heuristics. The basic idea here is to treat all clients in non-increasing order of their r_i values. For each client we identify the server with minimal available capacity that can treat all its requests (see Algorithm 14).

2.3.3 *Multiple*

6. Multiple Top Down (MTD) – The top-down approach for the *Multiple* policy is similar to UTD, with one significant difference: the delete-procedure (see Algorithm 15). For *Upwards*, requests of a client have to be treated by a single server, and it may occur that after the delete-procedure a server has still some capacity left to treat more requests, but all remaining clients have a higher amount of requests than this leftover capacity. For *Multiple*, requests of a client can be treated by multiple servers. So if at the end of the delete-procedure the server still has some capacity, we delete this amount of requests from the client with the largest r_i.

7. Multiple Bottom Up (MBU) – This heuristic is similar to MTD, except that we perform a bottom-up traversal of the tree in the first pass, and that the clients are deleted in non-decreasing order of their r_i-values. Algorithm 16 describes the first pass (servers with exhausted capacity). The second pass which adds extra servers if required is described in Algorithm 17.

8. Multiple Greedy (MG) – This last heuristic greedily allocates requests to servers in a bottom-up traversal of the tree, thus always finding a solution if there is one, but possibly at an expensive cost.

2.4 Experiments

We have done some experiments to assess the impact of the different access policies, and the performance of the polynomial heuristics described in Section 2.3. We obtain an absolute lower bound of the solution for each tree platform with the mixed integer linear program similar described in Section 2.2. We outline the experimental plan in Section 2.4.1. Results are given and commented in Section 2.4.2. In the following, we denote by s the problem size: $s = |\mathcal{C}| + |\mathcal{N}|$.

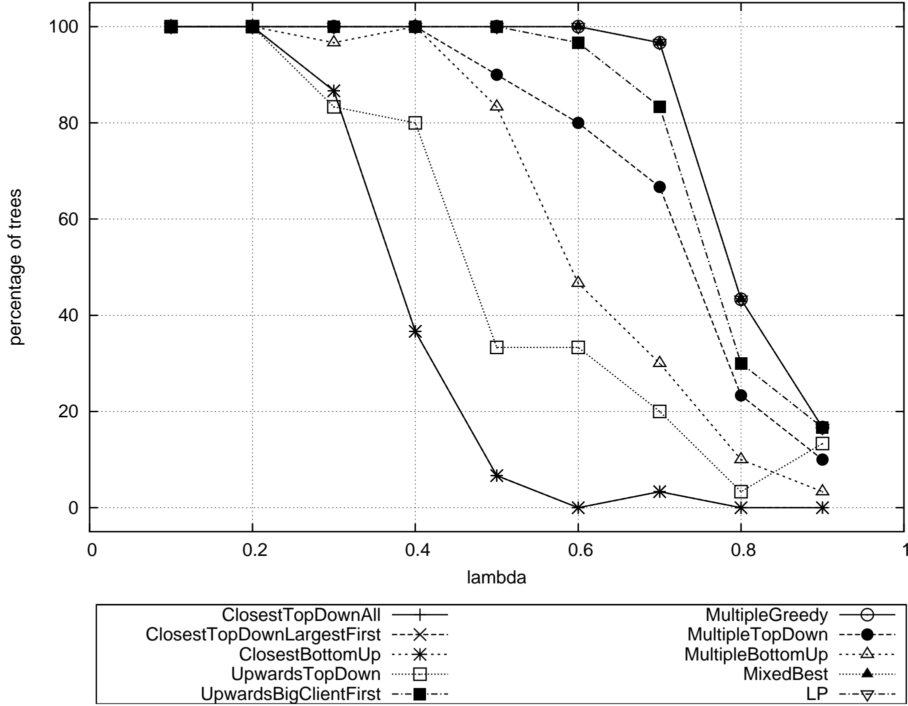


Figure 2.4: Homogeneous case - Percentage of success.

2.4.1 Experimental Plan

The important parameter in our tree networks is the load, i.e., the total number of requests compared to the total processing power: $\lambda = \frac{\sum_{i \in \mathcal{C}} r_i}{\sum_{j \in \mathcal{N}} \mathsf{W}_i}$. We have performed experiments on 30 trees for each of the nine values of λ selected ($\lambda = 0.1, 0.2, ..., 0.9$). The trees have been randomly generated, with a problem size $15 \leq s \leq 400$. When λ is small, the tree has a light request load, while large values of λ imply a heavy load on the servers. We then expect the problem to have a solution less frequently.

We have computed the number of solutions for each lambda and each heuristic. The number of solutions obtained by the linear program indicates which problems are solvable. Of course we cannot expect a result with our heuristics for the intractable problems.

To assess the performance of our heuristics, we have studied the relative performance of each heuristic compared to the lower bound. For each λ, results are computed on the trees for which the linear program has a solution. Let T_λ be the subset of trees with a solution. Then, the relative performance for the heuristic h is obtained by $\frac{1}{|T_\lambda|} \sum_{t \in T_\lambda} \frac{cost_{LP}(t)}{cost_h(t)}$, where $cost_{LP(t)}$ is the lower bound cost returned by the linear program on tree t, and $cost_h(t)$ is the cost involved by the solution proposed by heuristic h. In order to be fair versus heuristics who have a higher success rate, we set $cost_h(t) = +\infty$ if the heuristic did not find any solution.

Experiments have been conducted both on homogeneous networks (Replica Counting problem) and on heterogeneous ones (Replica Cost problem).

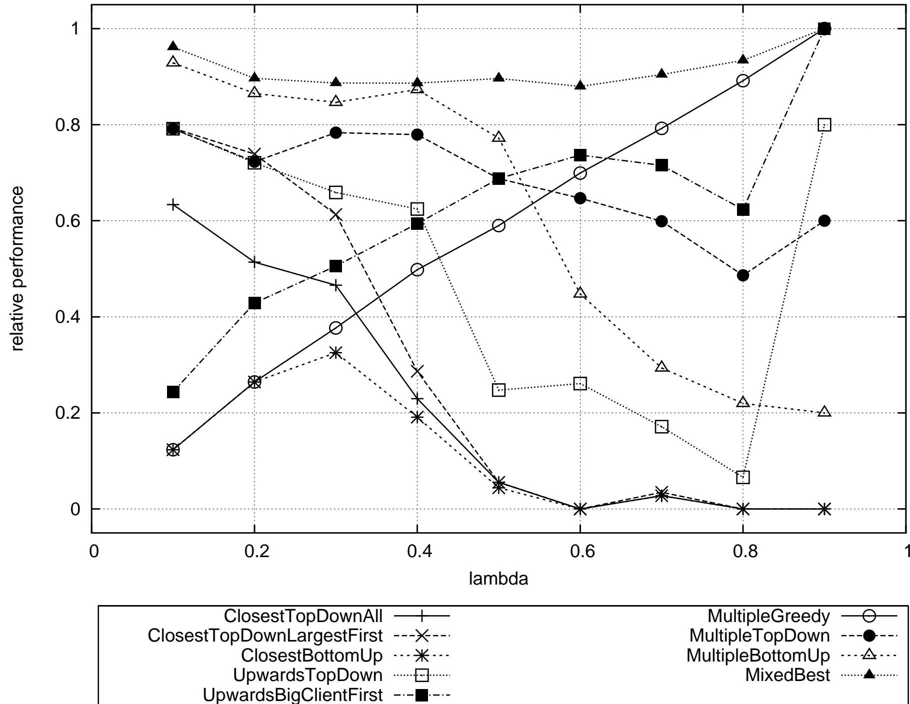


Figure 2.5: Homogeneous case - Relative performance.

2.4.2 Results

A solution computed by a *Closest* or *Upwards* heuristic always is a solution for the *Multiple* policy, since the latter is less constrained. Therefore, we can mix results into a new heuristic for the *Multiple* policy, called MixedBest (**MB**), which selects for each tree the best cost returned by the previous eight heuristics. Since MG never fails to find a solution if there is one, MB will neither fail either.

Figure 2.4 shows the percentage of success of each heuristic for homogeneous platforms. The upper curve corresponds to the result of the linear program, and to the cost of the MG and MB heuristics, which confirms that they always find a solution when there is one. The UBU heuristic seems very efficient, since it finds a solution more often than MTD and MBU, the other two *Multiple* policies. On the contrary, UTD, which works in a similar way to MTD and MBU, finds less solutions than these two heuristics, since it is further constrained by the *Upwards* policy. As expected, all the *Closest* heuristics find fewer solutions as soon as λ reaches higher values: the bottom curve of the plot corresponds to CTDA, CTDLF and CBU, which all find the same solutions. This is inherent to the limitation of the *Closest* policy: when the number of requests is high compared to the total processing power in the tree, there is little chance that a server can process all the requests coming from its subtree, and requests cannot traverse this server to be served by a server located higher in the tree. These results confirm that the new policies have a striking impact on the existence of a solution to the Replica Counting problem.

Figure 2.5 represents the relative performance of the heuristics compared to the LP-based lower bound. As expected, the hierarchy between the policies is respected, i.e., *Multiple* is better than *Upwards* which in turn is better than *Closest*. Altogether, the use of the MixedBest

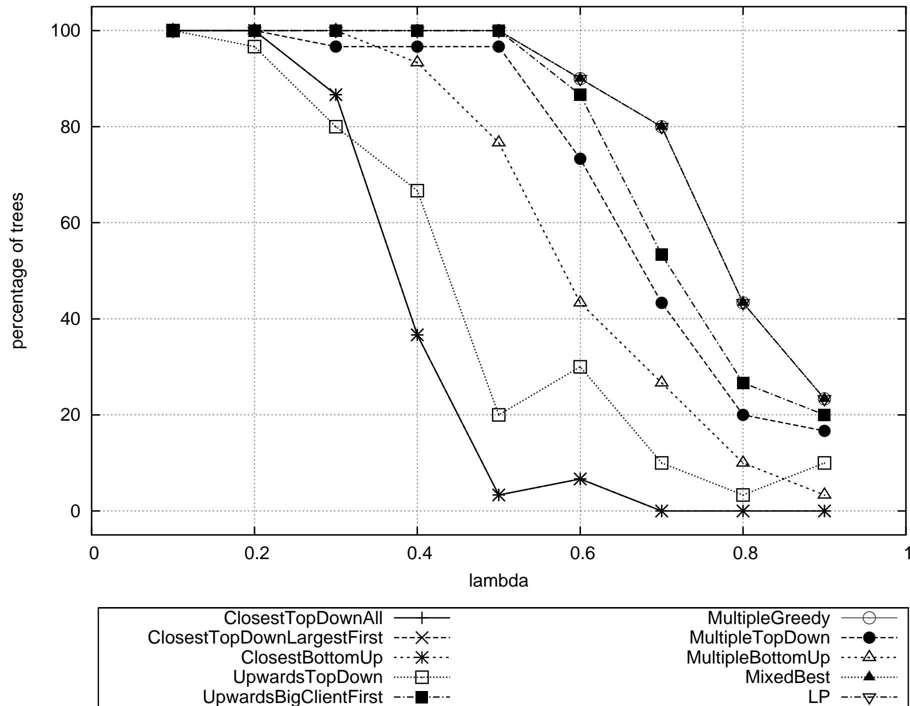


Figure 2.6: Heterogeneous case - Percentage of success.

heuristic MB allows to always pick up the best result, thereby resulting in a very satisfying relative cost for the *Multiple* instance of the problem. The greedy MG should not be used for small values of λ, but proves to be very efficient for large values, since it is the only heuristic to find a solution for such instances.

To conclude, we point out that MB always achieves a relative performance of at least 85%, thus returning a replica cost within 17% of that of the LP-based lower bound. This is a very satisfactory result for the absolute performance of our heuristics. Especially given the fact that running all heuristics takes less than a minute, while the LP solution requires up to several hours.

The heterogeneous results (see Figure 2.6 and Figure 2.7) are very similar to the homogeneous ones, which clearly shows that our heuristics are not much sensitive to the heterogeneity of the platform.

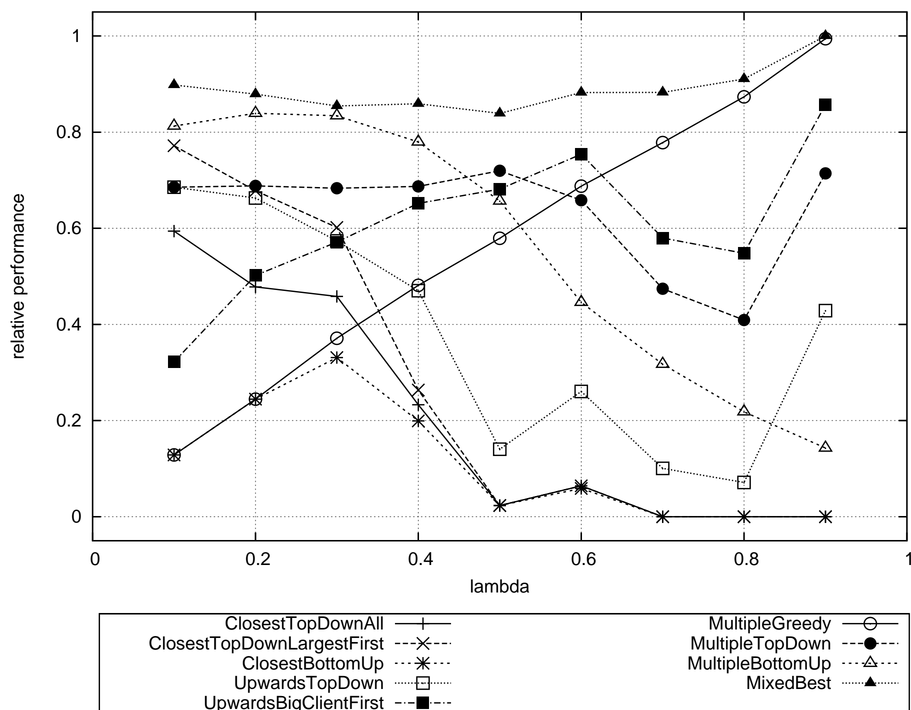


Figure 2.7: Heterogeneous case - Relative performance.

Chapter 3

Multi-Criteria Optimization Problems

This chapter deals with Replica Placement including more criteria in the objective function. More precisely, we study the problems Replica Placement with QoS and Replica Cost with QoS and Bandwidth. In Section 3.1 we present the residual complexity results for the different instances. The formulation of Replica Placement with QoS as linear program is subject of Section 3.2. We present several polynomial heuristics in Section 3.3 and the results of our experimentations are detailed in Section 3.4.

3.1 Complexity Results

Table 3.1 gives an overview of complexity results of the different instances of the Replica Counting problem (homogeneous servers). Bold items indicate those results that we are addressing in particular in this chapter. Liu et al. [46] provided a polynomial algorithm for the *Closest* policy with QoS constraints. In Chapter 2 we proved the NP-completeness of the *Upwards* policy without QoS. This was a surprising result, to be contrasted with the fact that the *Multiple* policy is polynomial under the same conditions (Cf. Chapter 2).

3.1.1 Replica Counting with QoS

An important contribution of this chapter is the NP-completeness of the *Multiple* policy with QoS constraints. As stated above, the same problem was polynomial without QoS, which gives a clear insight on the additional complexity introduced by QoS constraints.

Theorem 3.1. Replica Counting with QoS *and the Multiple strategy is NP-complete.*

Proof. The problem clearly belongs to the class NP: given a solution, it is easy to verify in polynomial time that all requests are served, that all QoS constraints are satisfied and that no server capacity is exceeded.

Table 3.1: Complexity results for the different instances of Replica Counting. Bold items denote the results treated in this chapter.

	Homogeneous	**Homogeneous/QoS**	**Hom./QoS/Bandwidth**
Closest	polynomial [22, 46]	polynomial [46]	**polynomial**
Upwards	NP-hard	NP-hard	NP-hard
Multiple	polynomial	**NP-hard**	NP-hard

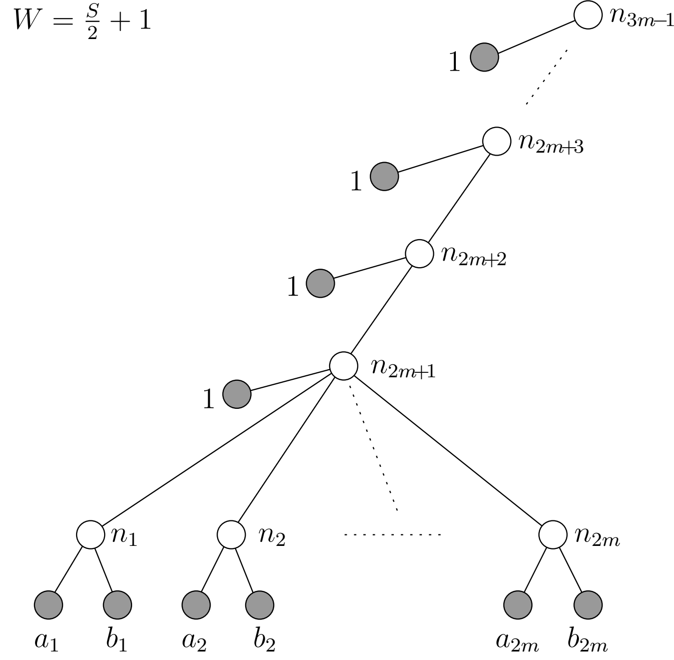


Figure 3.1: The platform used in the reduction for Theorem 3.1.

To establish the completeness, we use a reduction from 2-PARTITION-EQUAL [32]. We consider an instance $\mathcal{I}_1$ of 2-PARTITION-EQUAL: given $2m$ positive integers $a_1, a_2, \ldots, a_{2m}$, does there exist a subset $I \subset \{1, \ldots, 2m\}$ of cardinal m such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Let $S = \sum_{i=1}^{2m} a_i$, $W = \frac{S}{2}$ and $b_i = \frac{S}{2} - 2a_i$ for $1 \leq i \leq 2m$. We build the following instance $\mathcal{I}_2$ of our problem (see Figure 3.1):

- Problem size: there are $5m - 1$ clients c_i and $3m - 1$ internal nodes n_j:
- Nodes: for $1 \leq j \leq 2m$, node n_j has capacity $\mathsf{sc}_j = \mathsf{W}_j = W$
  - For $1 \leq j \leq 2m$, the parent of node n_j is node n_{2m+1}
  - For $2m + 1 \leq j \leq 3m - 2$, the parent of node n_j is node n_{j+1}
  - Node n_{3m-1} is the root r of the tree.
- Clients:
  - For $1 \leq i \leq 2m$, client c_i has $r_i = a_i$ requests of QoS $q_i = 2$, and its parent is node n_i
  - For $2m + 1 \leq i \leq 4m$, client c_i has $r_i = b_{i-2m}$ requests of QoS $q_i = m$, and its parent is node n_{i-2m}
  - For $4m + 1 \leq i \leq 5m - 1$, client c_i has $r_i = 1$ request of QoS $q_i = 1$ and its parent is node n_{i-2m}.

Finally, we ask whether there exists a solution with total storage cost $(2m - 1)W$, i.e., with $2m - 1$ servers. Clearly, the size of $\mathcal{I}_2$ is polynomial (and even linear) in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. We assign a replica to each node n_i, $i \in I$ (by hypothesis there are m of them), and one in each of the $m - 1$ top nodes n_{2m+1} to n_{3m-1}. All $m - 1$ clients with QoS 1 are served by their parent.

For $1 \leq i \leq 2m$ there are tow cases:
- If $i \in \mathcal{I}$, both clients c_i and c_{i+2m} are served by their parent n_i. Node n_i serves a total of $a_i + b_i = \frac{S}{2} - a_i \leq W$ requests.
- If $i \notin \mathcal{I}$, client c_i is served by node n_{2m+1} and client c_{i+2m} is served by one or several ancestors of n_{2m+1}, i.e., nodes n_{2m+2} to n_{3m-1}. Node n_{2m+1}, which also serves the unique request of client c_{2m+1}, serves a total of $\sum_{i \notin I} a_i + 1 = W$ requests. The $m-2$ ancestors of n_{2m+1} receive the load $\sum_{i \notin I} b_i = mS - 2S$. They also serve $m-2$ clients with a single request, hence a a total load of $(m-2)S + m - 2 = (m-2)W$ requests to distribute among them. This is precisely the sum of their capacities, and any assignment will do the job.
Note that the allocation of requests to servers is compatible with all QoS constraints. All requests with QoS 1 are served by the parent node. All requests with QoS 2, i.e., with value a_i, are served either by the parent node (if $i \in I$) or by the grandparent node (if $i \notin I$). Altogether, we have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution with $2m-1$ servers. Necessarily, there is a replica located in each of the top $m-1$ nodes n_{2m+1} to n_{3m-1}, otherwise some request with QoS 1 would not be served satisfactorily. Each of these nodes serves one of these requests, hence has remaining capacity $W - 1 = \frac{S}{2}$.

There remain m servers which are placed among nodes n_1 to n_{2m}. Let I be the set of indices of those m nodes which have not received a replica. Necessarily, requests a_i, with $i \in I$, are served by node n_{2m+1}, because of their QoS constraint. Hence $\sum_{i \in I} a_i \leq \frac{S}{2}$. Next, all requests a_i and b_i, with $i \in I$, are served by nodes n_{2m+1} to n_{3m-1}, whose total remaining capacity is $(m-1)\frac{S}{2}$. There are $(\sum_{i \in I} a_i) + (m\frac{S}{2} - 2\sum_{i \in I} a_i)$ such requests, hence

$$m\frac{S}{2} - \sum_{i \in I} a_i \leq (m-1)\frac{S}{2}.$$

From this equation we derive that $\sum_{i \in I} a_i \geq \frac{S}{2}$. Finally we have $\sum_{i \in I} a_i = \frac{S}{2}$, with $|I| = m$, hence a solution to $\mathcal{I}_2$.

■

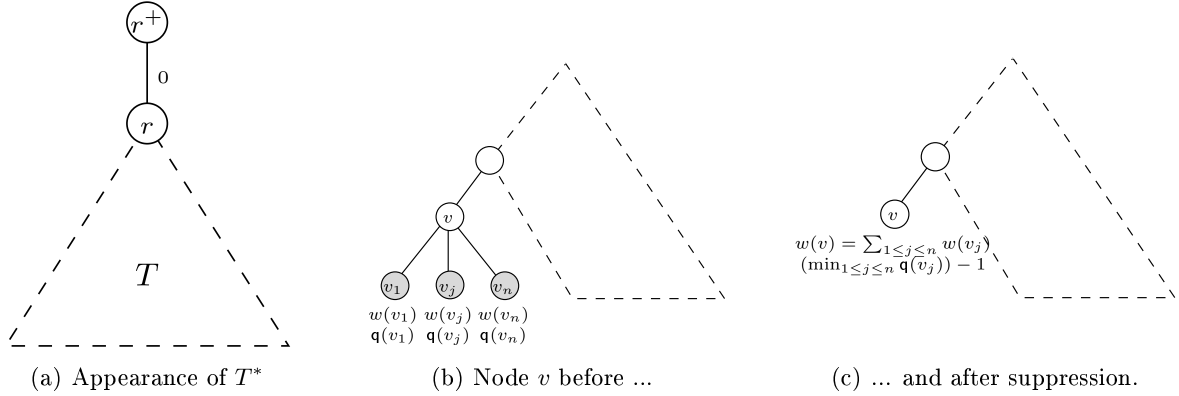
Remember that all three instances of the Replica Placement problem (heterogeneous servers with the *Closest*, *Upwards* and *Multiple* policies) are already NP-complete without QoS constraints (Cf. Chapter 2).

3.1.2 Replica Cost with QoS and Bandwidth

Theorem 3.2. *The instance of* Replica Cost with QoS and Bandwidth *with the Closest strategy can be solved in polynomial time.*

Proof. We outline below an optimal algorithm, called *ORP*, to solve the problem. ■

For our algorithm, we modify an optimal algorithm of Lin, Liu and Wu [46] for Replica Cost with QoS with the *Closest* policy. To be able to use their algorithm **Place-replica**, we have to modify the original platform: we transform the distribution tree T in a tree T^* by adding a new root r^+ as father of the original root r (see Figure 3.2(a)). r^+ is connected to r via a link l_0, where $\mathsf{BW}(l_0) = 0$. As the bandwidth is limited to 0, no requests can pass above r, so that this artificial transformation for computation purposes can be adapted to any tree-network. We make this changing to be able to model whether the original root r is equipped with a replica or not.



(a) Appearance of T^* (b) Node v before ... (c) ... and after suppression.

Figure 3.2: Transformations.

A further, only formal transformation, consists in the suppression of clients from the tree and hence the consideration of their parents as leaves in the following way: for every parent p who has only leaf-children $v_1, .., v_n$, we assign the sum of the requests of the v_j as its requests $w(p)$, i.e., $w(p) = \sum_{1\leq j\leq n} w(v_j)$. The associated QoS is set to $(\min_{1\leq j\leq n} \mathsf{q}(v_j)) - 1$. (Figures 3.2(b) and 3.2(c) give an illustration). This transformation is possible, as we use the *Closest* policy and hence all children have to be treated by the same server. From those parents who have some leaf-children $v_1, .., v_n$, but also non-leaf children $v_{n+1}, .., v_m$, the clients can not be suppressed completely. In this case the leaf-children $v_1, .., v_n$ are compressed to one single client c with requests $w(c) = \sum_{1\leq j\leq n} w(v_j)$ and QoS $\mathsf{q}(c) = \min_{1\leq j\leq n} \mathsf{q}(v_j)$. Once again this compression is possible due to the restriction on the *Closest* access policy.

ORP works in two phases. In the first phase so called *Contribution Functions* are computed which will serve in the second phase to determine the optimal replica placements. In the following some new terms are introduced and then the two phases are described in detail.

Terminology

Working with a tree T^* with root r^+, we note $t(v)$ the subtree rooted by node v, and $t'(v) = t(v) - v$, i.e., the forest of trees rooted at v's children. The i-th ancestor of node v, traversing the tree up to the root, is denoted by $a(v, i)$. See Figure 3.3 for an illustration.

Using these notations, we denote $m(T^*)$ the minimum cardinality set of replicas that has to be placed in tree T such that all requests can be treated by a maximum processing capacity of W (respecting QoS and bandwidth constraints). In the same manner $m(t(v))$ denotes the minimum number of replicas that has to be placed in $t'(v)$, such that the remaining requests on node v are within W. For this purpose we define a contribution function C. $C(v, i)$ denotes the minimum number of requests on node $a(v, i)$ contributed by $t(v)$ by placing $m(t(v))$ replicas in $t'(v)$ and none on $a(v, j)$ for $0 \leq j < i$. The computation is presented below (Cf. Section 3.2). But before we need a last notation. The set $e(v, i)$ denotes the children of node v that have to be equipped with a replica such that the remaining requests on node $a(v, i)$ are within W, there are exactly $m(t(v))$ replicas in $t'(v)$ and none on $a(v, j)$ for $0 \leq j < i$ and the contribution $t(v)$ on $a(v, i)$ is minimized. The computation formula is also given below.

Phase 1: Bottom up computation of set e, amount m and contribution function C

The computation of e, m and C is a bottom up process, distinguishing two cases.

1. v is a leaf In this case we do not need e and m and we can directly compute the contribution function. $C(v,i)$ is $w(v)$ when $(i \leq \mathsf{q}(v) \wedge w(v) \leq \min_{\mathsf{BW}} \mathsf{path}[v \rightarrow a(v,i)])$, and infinity otherwise.

We point out that there is no solution if any of the leaves has more requests than W or if the bandwidth of any of the clients to its parent is not sufficiently high.

2. v is an internal node with children $v_1, \ldots, v_n$
$i = 0$: If the contribution on v of its children, i.e., the incoming requests on v is bigger than the processing capacity of inner nodes W, we know we have to place some replicas on the children to bound the incoming requests on W. To find out which children have to be equipped with a replica, we take a look at the $C(v_j, 1)$-values of the children. The set $e(v,0)$ is used to store the v_j's that are determined to be equipped with a replica. Hence the procedure is the following:

$e(v,0) = \emptyset$;
while $\sum_{v_j \notin e(v,0)} C(v_j,1) > W$ **do**
  add $v_j \in \mathcal{N}$ with biggest $C(v_j,1)$ to $e(v,0)$;

Note that the set $\mathcal{N}$ used in the procedure still corresponds to the set of internal nodes of the original tree T. So we can add leaf nodes of T^* that are inner nodes in T, but we can not add compressed client nodes. Note furthermore that there is no client that is added to $e(v,0)$. Besides we remark that there is no valid solution within W and the present QoS and bandwidth constraints, when all children $v_j \in \mathcal{N}$ of v are equipped with a replica and the incoming requests do not fit in W. Of course this holds also true in the case $i > 0$. Subsequently, the value of $m(t(v))$ is determined easily: $m(t(v)) = \sum_{1 \leq j \leq n} m(t(v_j)) + |e(v,0)|$. We remind that $m(t(v))$ indicates the minimum number of replicas that have to be placed in $t'(v)$ to keep the number of contributed requests inferior to W. Finally, the computation of the contribution function :

$$C(v,0) = \sum_{v_j \notin e(v,0)} C(v_j,1).$$

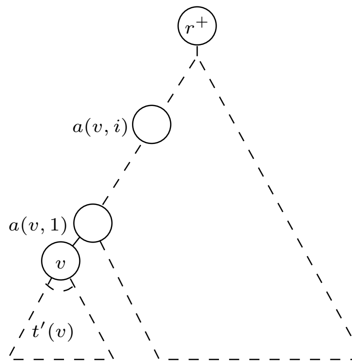


Figure 3.3: Clarification of the terminology.

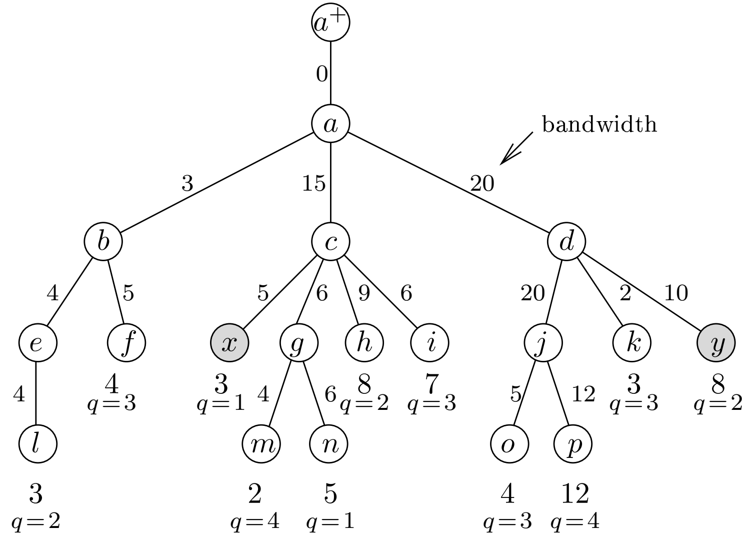


Figure 3.4: Example

$i > 0$: Treating node v, we want to compute the contribution on $a(v, i)$. As for $i = 0$, we start computing the set $e(v, i)$:

$e(v, i) = \emptyset$;
while $\sum_{v_j \notin e(v,i)} C(v_j, i+1) > W$ **do**
  add $v_j \in \mathcal{N}$ with biggest $C(v_j, i+1)$ to $e(v, i)$;

The computation of the contribution function follows a similar principle:

$$C(v, i) = \begin{cases} \sum_{v_j \notin e(v,i)} C(v_j, i+1), & \text{if } |e(v, i)| = |e(v, 0)| \\ \infty, & \text{otherwise} \end{cases} \tag{3.1}$$

$C(v, i)$ is set to ∞, when the number of $|e(v, 0)|$ replicas placed among the children of v is not sufficient to keep the contributed requests on $a(v, i)$ within W.

Example of Phase 1 Consider the tree in Figure 3.4 and a processing capacity of inner nodes fixed to $W = 15$. The tree has already been transformed. So nodes x and y are compressed client-leaves (grey scaled in the figure), whereas all other leaves correspond to servers (former inner nodes, hence nodes that are within $\mathcal{N}$). We start with the computation of all $C(v, i)$-values of all leaves. Leaf l for example has $C(l, 0) = 3$ as it holds 3 requests. As the link from l to e has a bandwidth of 4, and the QoS is 2, the requests of l can ascent to node e and hence the contribution of l's requests on node e, $C(l, 1)$, is 3. In the same manner, $C(l, 2)$, i.e., the contribution of l's requests on node b is 3 as well. But then the QoS range is exceeded and hence the requests of l can not be treated higher in the tree. Consequently the contributions on nodes a and a^+ ($C(l, 3)$ and $C(l, 4)$) are set to infinity.

Table 3.3 is used for the computation of e, m and C values of inner nodes. During the computation process it is filled by main columns, where one main column consists of all inner nodes of the same level in the tree. So we start with node e. The contribution of its child

	l	f	x	m	n	h	i	o	p	k	y
$C(v,0)$	3	4	3	2	5	8	7	4	12	3	8
$C(v,1)$	3	4	3	2	5	8	∞	4	12	∞	8
$C(v,2)$	3	∞	∞	2	∞	8	∞	4	12	∞	8
$C(v,3)$	∞	∞	∞	2	∞	∞	∞	4	12	∞	∞
$C(v,4)$	∞			∞	∞			∞	∞		

Table 3.2: Computation of $C(v,i)$-values of leaves.

	e	g	j	b	c	d	a	a$^+$
$e(v,0)$	$\emptyset$	$\emptyset$	$\{p\}$	$\emptyset$	$\{g,i\}$	$\{k\}$	$\{b,c\}$	$\{a\}$
$m(t(v))$	0	0	1	0	2	2	6	7
$C(v,0)$	3	7	4	9	11	12	12	∞
$e(v,1)$	$\emptyset$	$\{n\}$	$\{p\}$	$\{e\}$	$\{g,i\}$	$\{k\}$	$\{b,c,d\}$	
$C(v,1)$	3	∞	4	∞	∞	12	∞	
$e(v,2)$	$\{l\}$	$\{n\}$	$\{p\}$	$\{e,f\}$	$\{g,i\}$	$\{j,k\}$		
$C(v,1)$	∞	∞	4	∞	∞	∞		
$e(v,3)$	$\{l\}$	$\{m,n\}$	$\{o,p\}$					
$C(v,1)$	∞	∞	∞					

Table 3.3: Computation of e, m and C for internal nodes.

l, $C(l,1)$, is 3. As it is the only child, we have that the contributed requests on e are less than the processing capacity $W = 15$ and hence we do not need to place a replica on its child l. Corresponding we get $m(t(e)) = 0$ and a contribution $C(e,0) = 3$. $e(e,1)$ and $C(e,1)$ are computed in the same manner, taking into account $C(l,2)$. Computing $e(e,2)$, i.e., the nodes that have to be equipped with a replica if we want to minimize the contribution on node $a(e,2) = a$ by placing replicas on the children of e but none on e up to a. For this purpose we use $C(l,3)$, the contribution of l on a and remark that it is infinity. Hence we have to equip l with a replica, and as now the set $e(e,2)$ has a higher cardinality than $e(e,0)$, we know that this solution is not optimal anymore and we set the contribution of $C(e,2)$ to infinity (Eq. 3.1). Taking a look at node j: In the computation of $e(j,0)$, we have a total contribution of its children of 16, which exceeds the processing power of $W = 15$ (bandwidth and QoS are not restricting here). Indeed we have to equip one of the children with a replica, and we choose the one with the highest contribution on j: node p. Consequently, we get $m(t(j)) = 1$ as we have to place one replica on the children. The contribution $C(j,0)$ consists in the 4 remaining contributed requests of node o. Once we have finished all computations for this level, we start with the computations of the next level, which can be found in the next main column of the table.

Phase 2: Top down replica placement

The second phase uses the precomputed results of the first phase to decide about the nodes on which to place a replica. The goal is to place $m(T^*) = m(t(r^+))$ replicas in $t'(r^+)$. Note that this means that there is no replica on r^+ and hence only the original tree T will be equipped with replicas. If the number of contributed requests on node r is within W, we have a feasible solution.

Phase 2 is a recursive approach. Starting with $i = 0$ on node $v = r^+$, all nodes that are

within $e(v, i)$ are equipped with a replica. In this top down approach, i indicates the distance of node v to its first ancestor up in the tree that is equipped with a replica and hence the set $e(v, i)$ denotes the set of children of v that have to be equipped with a replica in order to minimize the contribution of v on $a(v, i)$. Next the procedure is called recursively with the appropriate index i. Algorithm 4 gives the pseudo-code for the top down placement phase, which is the same as the one in [46].

Algorithm 4: Top down replica placement

```
procedure Place-replica (v, i)
if v ∈ C then
  return
place a replica at each node of e(v, i);
for all c ∈ children(v) do
  if c ∈ e(v, i) then
    Place-replica(c,0);
  else
    Place-replica(c,i+1);
```

Example of Phase 2 We start with the results of Phase 1 (Cf. Table 3.2 and 3.3) and call then the procedure Place-replica (Algorithm 4) with $(a^+, 0)$. a^+ is not a leaf, so we place a replica on its child a, as $a \in e(a, 0)$ and then recall the procedure with $(a, 0)$. This time we place replicas on b and c and call the procedure with values $(b, 0), (c, 0)$ and $(d, 1)$. We have to increment i to 1 when we treat node d, as we already know that we will not equip d with a replica, and hence the children of d might give their contribution directly to a. So we have to examine which of the children of d have to be equipped with a replica, to minimize the contribution on a. This is stored in the $e(v_j, 1)$-values of all children v_j of d. So every time we do not place a replica on a node and descent to its children, we increase the distance-indicator i to the first replica that can be found the way up to the root. At the end we get this set of replicas: $R = \{a, b, c, g, i, k, p\}$.

The recursive procedure call for the entire example is given in Table 3.1.2. PR(x,i) stands for the call of Place-replica with parameters (x,i) and $\rightarrow x$ indicates that node x is equipped with a replica.

Complexity of *ORP*

For each node v we have to compute e, m and C values. So the computation requires $n \log n$, if v has n children and if we sort the C values from all of v's children. We have to do at most L sorting, where L is the maximum range limit among all nodes. So at all the computation complexity for the values for one node is $Ln \log n$, and we get a total complexity of $LN \log N$, where N is the number of nodes in the tree.

Proof of Optimality

To prove optimality of our algorithm *ORP*, we use a recursion over levels. For this purpose we apply a theorem introduced by Liu et al. [46] and presented below as Theorem 3.3. Liu et al. used this theorem in order to prove the existence of an optimal solution on a homogeneous data

PR(a$^+$,0)								
$\rightarrow$ **a**								
PR(a,0)								
$\rightarrow$ **b,c**								
PR(b,0)		PR(c,0)				PR(d,1)		
		$\rightarrow$ **g,i**				$\rightarrow$ **k**		
PR(e,1)	PR(f,1)	PR(g,0)		PR(h,2)	PR(i,0)	PR(j,2)		PR(k,0)
	ret			ret	ret	$\rightarrow$ **p**		ret
PR(l,2)		PR(m,1)	PR(n,0)			PR(o,3)	PR(p,0)	
ret		ret	ret			ret	ret	

Table 3.4: Scheme on the recursive calls of the procedure **Place-replica**

grid tree under QoS constraints. As the theorem does not take into account if there are any constraints like QoS or bandwidth, we can adopt it for our problem.

Theorem 3.3. *Consider a data grid tree T, a node v in T with children $v_1, .., v_n$ and a workload W. There exists a replica set R so that $|R| = m(T)$, R minimizes the total workload due to R from $t'(v)$ on $a(v, i)$ for $i \geq 1$, and $|R \cap t'(v_j)| = m(t(v_j))$.*

In other words, Theorem 3.3 guarantees that for a tree T with fixed processing capacity W there exists a replica set R whose cardinality is the minimum number of replicas that has to be placed in $t'(r)$ (where r is the root of T), such that the remaining requests on r are within W. Furthermore for a node v with children $v_1, ..., v_n$, due to R the workload on a ancestor $a(v, i)$ of v is minimized and the number of replicas that are placed in the subtree $t'(v_j)$ is minimal.

Proof. We can use the same arguments as Liu et al. as we did not change the definition of m-values but the constraints on m. By definition of $m(t(v_j))$, we know that this is the minimal number of replicas that has to be placed in $t'(v_j)$ such that the contribution on v_j is within W. Hence $|R \cap t'(v_j)$ can not be less than $m(t(v_j))$ because otherwise the contribution on v_j would exceed W. On the other side in any optimal solution for $t(v_j)$, we can not place more replicas in $t'(v_j)$ than $m(t(v_j))$ and than one more on v_j. The resulting contribution on $a(v, i)$ decreases at most when placing the replica on v_j. ∎

Theorem 3.4. *Algorithm ORP returns an optimal solution to the* Replica Placement *problem with fixed W, QoS and bandwidth constraints, if there exists a solution.*

Proof. We perform an induction over levels to prove optimality. We consider any tree T^* of height $n + 1$ and start at level 0, which consists in the artificial root r^+ (Cf. Figure 3.5).

level 0: Using Theorem 3.3, we know that there exists an optimal solution R_0 for our tree (i.e., a set R of replicas whose cardinality is $m(T^*)$) such that $|R_0 \cap t'(r)| = m(t(r))$. We have $m(T^*) = m(t(r)) + |e(r^+, 0)|$ by definition of $e(r^+, 0)$. Hence $e(r^+, 0) = \{r\}$ if and only if $r \in R_0$. This is exactly how the algorithm pursuits.

level i $\rightarrow$ i+1: We assume that we have placed the replicas from level 0 to level i (with Algorithm 4) and that there exists an optimal solution R_i with these replicas. We further suppose that for each node v in level i it holds $|R_i \cap t'(v)| = m(t(v))$. Let us consider a node v in level i with children $v_1, .., v_n$ and we define $l := \min\{k \geq 0 | a(v, k) \in R_i\}$. In

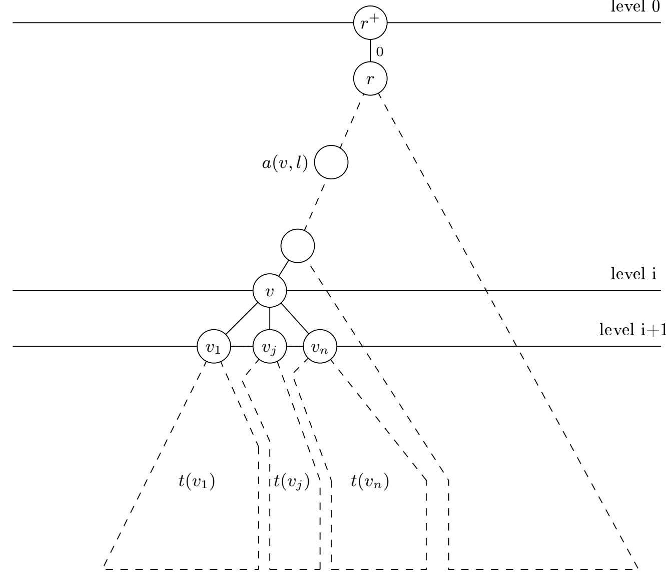


Figure 3.5: Induction over levels.

the next step of the algorithm we equip the elements of $e(v,l)$ with a replica. We have $m(t(v)) = \sum_{1 \leq j \leq n} m(t(v_j)) + |e(v,0)|$, i.e., the minimal number of replicas in the subtrees $t'(v_j)$ and the minimal number of replicas on the children of v that have to be placed to keep the contributed requests on v within W. By definition of $e(v,l)$ we have that $|\{j \in \{1,..,n\}|v_j \in R_i\}| \geq |e(v,l)|$ and we also have $|e(v,l)| = |e(v,0)|$ as the contribution $C(v,l)$ is finite and R_i a solution. For the inequality, there is even equality because otherwise there would exist a j such that $|t'(v_j) \cap R_i| < m(t(v_j))$, which is impossible. With this equality, we can replace the children of v that are in R_i by the children of v that are in $e(v,l)$ creating a solution R_{i+1}. So R_{i+1} is also an optimal solution, because $|R_i| = |R_{i+1}|$ (we did not change the nodes of the other levels) and the contribution of $t(v)$ on $a(v,l)$ has at most decreased. Furthermore for every node v' at level i+1 we have $|R_{i+1} \cap t'(v')| = m(t(v'))$.

So the last solution R_n that we get in the induction step n is optimal and it corresponds to the solution that we obtain by our algorithm. ■

3.2 Linear Programming Formulation for Replica Placement with QoS

In this section we describe how to extend the mono-criteria linear programs presented in Section 2.2 to Replica Placement with QoS. The core keeps the same, we only need to add the additional constraints in the objective functions.

3.2.1 Extension of the Mono-Criteria Linear Program

Single server

For the single server strategies, *Closest* and *Upwards*, we have to add the following constraints ensuring that QoS claims are respected:

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i), \mathsf{dist}(i,j) y_{i,j} \leq \mathsf{q}_i,$$

where $\mathsf{dist}(i,j) = |\mathsf{path}[i \to j]|$, the number of links needed to reach j from i. As stated previously, we could take the computational time of a request into account by writing $(\mathsf{dist}(i,j) + \mathsf{comp}_j) y_{i,j} \leq \mathsf{q}_i$, where comp_j would be the time to process a request on server j.

Multiple servers

In the case of the *Multiple* policy, the additional constraint for QoS requirements is the following: QoS:

$$\forall i \in \mathcal{C}, \forall j \in \mathsf{Ancestors}(i), \mathsf{dist}(i,j) y_{i,j} \leq \mathsf{q}_i y_{i,j}$$

Note that it is also possible to further extend the linear programs to bandwidth constraints. In this case we have to ensure that the bandwidth of any link cannot be exceeded. This can be expressed as $\forall l \in \mathcal{L}, \sum_{i \in \mathcal{C}} r_i z_{i,l} \leq \mathsf{BW}_l$ in the single server case, and as $\forall l \in \mathcal{L}, \sum_{i \in \mathcal{C}} z_{i,l} \leq \mathsf{BW}_l$ for multiple servers.

3.2.2 An Exact MIP-Based Solution for *Multiple*

In this section we prove that we are able to compute an optimal solution for Replica Placement with QoS for *Multiple* using a mixed integer version of our linear program (MIP).

Theorem 3.5. *The solution of the linear program detailed in Sections 2.2 and 3.2.1, when solved with all variables being rationals except of the x_i, is an achievable bound for the Multiple problem with QoS constraints, and we can build an exact solution in polynomial time, based on the LP solution.*

Proof. Consider the solution of the LP program:

- $\forall i \in \mathcal{C},\ x_i \in \{0,1\}$
- $\forall i \in \mathcal{C}, \forall j \in \mathcal{N},\ y_{i,j} \in \mathbb{Q}$
- $\forall i \in \mathcal{C}, \forall l \in \mathcal{L},\ z_{i,l} \in \mathbb{Q}$

To prove that the lower bound obtained by this program is achievable, we are building an integer solution where $y'_{i,j}$ and $z'_{i,l}$ are integer numbers, keeping the same x_i and without breaking any constraints.

In the following, for any variable y, $\lfloor y \rfloor$ is the integer part of y, and $\tilde{y}$ the fractional part: $y = \lfloor y \rfloor + \tilde{y}$, and $\tilde{y} < 1$.

Let us consider a client $i \in \mathcal{C}$ such that $\exists j \in \mathcal{N} \mid \tilde{y}_{i,j} > 0$, i.e., $y_{i,j}$ is not an integer. We consider j_1 being the closest server to i not serving an integer number of requests of client i, and more generally $j_k, k = 1..K$ the servers on the path from i to the root, such that $\tilde{y}_{i,j_k} > 0$. We want to move bits of requests in order to obtain an integer value for y_{i,j_1}. This elementary transformation is called $\mathsf{trans}(i, j_1)$. We consider the two following cases.

First case:

$$\sum_{i' \in \mathsf{subtree}(j_1) \cap \mathcal{C}} y_{i',j_1} \leq W_{j_1} - (1 - \tilde{y}_{i,j_1})$$

In this case, there is enough space at server j_1 to fulfill an integer number of requests from client i. Since the total number of requests of client i is an integer, $\sum_{k=1}^{K} \tilde{y}_{i,j_k}$ is a non null integer. Thus, $\sum_{k=2}^{K} \tilde{y}_{i,j_k} \geq 1 - \tilde{y}_{i,j_1}$, and we can move down $1 - \tilde{y}_{i,j_1}$ bits of requests from servers $j_k, k = 2..K$ to j_1. No constraints will be violated since there is enough space on the server. The move is done by changing the values of y_{i,j_k} and recalculating the $z_{i,l}$ for $l \in \mathsf{path}[i \to r]$. After such a transformation, y_{i,j_1} is an integer variable.

Second case: If server j_1 is already too full in order to add a fraction of requests from client i, we need to exchange some requests with other clients. First, if there is some free space on the server, we start by filling completely server j_1 with fractions of requests of client i from servers j_k, $k = 2..K$. We know there are such requests, otherwise y_{i,j_1} would be an integer. This transformation is similar as the one done in the first case. We now have $\sum_{i' \in \mathsf{subtree}(j_1) \cap \mathcal{C}} y_{i',j_1} = W_{j_1}$. Let us denote by $i_t, t = 1..T$ the clients $i_t \in \mathsf{subtree}(j_1) \cap \mathcal{C} \setminus \{i\}$ such that $\tilde{y}_{i_t,j_1} > 0$. Since W_{j_1} is an integer and $\tilde{y}_{i,j_1} > 0$, we have $\sum_{t=1}^{T} \tilde{y}_{i_t,j_1} \geq 1 - \tilde{y}_{i,j_1}$, and also $\sum_{k=2}^{K} \tilde{y}_{i,j_k} \geq 1 - \tilde{y}_{i,j_1}$. We can select in both sets $1 - \tilde{y}_{i,j_1}$ bits of requests which will be exchanged, i.e., bits of requests from client i_t initially treated by j_1 will be moved on some servers j_k, which are in $\mathsf{Ancestors}(j_1)$, and the corresponding amount of requests of i will be moved back on server j_1.
In this case, we may break a QoS constraint since it is not sure that clients i_t can be served higher than j_1 in order to respect their QoS. However, we will see that in the general transformation process, we prevent such cases to happen. Note that all other constraints are still fulfilled. but just change the origin of these requests.

Once $\mathrm{trans}(i, j_1)$ has been done, y_{i,j_1} is an integer, and notice that only non-integer bits of requests have been moved, so we have not affected any integer part of the solution and we have decreased at least by one the number of non-integer variables in the solution.

Let us detail now the complete transformation algorithm, in order to obtain an integer solution. Particular attention must be paid to respect the QoS at all time.

```
for j ∈ N taken in a bottom-up traversal order do
    finish=1;
    while (finish==1) do
        C' = {i' ∈ C ∩ subtree(j) | ỹ_{i',j} > 0};
        if C' == ∅ then finish=0; else
            i = Min_{i'∈C'} (q_{i'} − dist(i', j));
            trans(i, j);
        end
    end
end
```

We consider each server in a bottom-up order, so that we are sure that each time we perform an elementary transformation, the server is the first one on the way from the client to the root having a non integer number of requests. In fact, when transforming server j, each server in $\mathsf{subtree}(j)$ has already been transformed, and thus have no fraction numbers of requests.

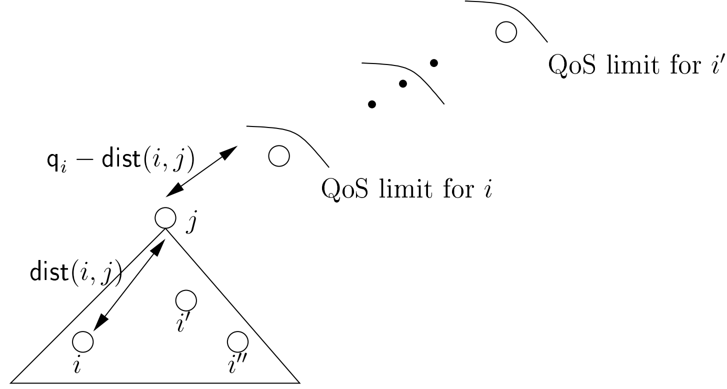


Figure 3.6: Illustration of the transformation algorithm

In order to transform server j, we look at the set $\mathcal{C}'$ of clients having a non-integer number of requests processed at j. If the set is empty, there is nothing to transform at j. Otherwise, we perform the elementary transformation with the client i which minimizes $(\mathsf{q}_{i'} - \mathsf{dist}(i', j))$, for $i' \in \mathcal{C}'$. This ensures that when we perform an elementary transformation as in the second case above, the QoS constraint will be respected for all clients i_t, since we are moving their requests into servers at distance at most $d = \mathsf{q}_i - \mathsf{dist}(i, j)$ from j, and their own QoS allows them to be processed at a distance $\mathsf{q}_{i_t} - \mathsf{dist}(i_t, j) \geq d$. Figure 3.6 illustrates this phase of the algorithm.

At the end of the while loop, server j is processing only integer numbers of requests, and thus we will not modify its requests affectation any more in the following.

The constraints are all respected at all step of the transformation, and we do not add or remove any replica, so the solution has exactly the same cost than the initial LP-based solution, and the transformed solution is fully integer. Moreover, this transformation algorithm works in polynomial time, in the worst case in $|\mathcal{N}| + |\mathcal{C}|^2$ but most of the time it is much faster since the transformations do not concern all clients simultaneously but only a few of them.

■

3.3 Heuristics for the Replica Placement Problem with QoS Constraints

In this section several heuristics including QoS constraints for the *Closest*, *Upwards* and *Multiple* policies are presented[1]. As already pointed out, the quality of service is the number of hops that requests of a client are allowed to traverse until they have to reach their server. All heuristics described below have polynomial, and even worst-case quadratic, complexity $O(s^2)$, where $s = |\mathcal{C}| + |\mathcal{N}|$ is the problem size.

In the following, we denote by $\mathsf{inreqQoS}_j$ the amount of requests that reach an inner node j within their QoS constraints, and by inreq_j the total amount of requests that reach j (including requests whose QoS constraints are violated).

[1] To ensure the reproducibility of our results, the code for all heuristics is available on the web: http://graal.ens-lyon.fr/~vsonigo/code/replicaQoS/

3.3.1 *Closest*

Closest Big Subtree First - CBS. Here we traverse the tree in top-down manner. We place a replica on an inner node j if $\mathsf{inreqQoS}_j \leq \mathsf{W}_j$. When the condition holds, we do not process any other subtree of j. If this condition does not hold, we process the subtrees of j in non-increasing order of inreq_j. Once no further replica can be added, we repeat the procedure. We stop when no new replica is added during a pass.

Closest Small QoS First - CSQoS. This heuristic uses a different approach. We do not execute a tree traversal. Instead, we sort all clients by non-decreasing order of q_i. In case of tie, clients are sorted by non-increasing order of r_i. For each client, we look for the server that can process its subtree ($\mathsf{inreqQoS}_j \leq \mathsf{W}_j$) and which is the nearest to the root. If no server is found for a client, we continue with the next client in the list. Once we reach a client in the list that is already treated by an earlier chosen server, we delete all treated clients from the to-do list and restart at the beginning of the remaining client list. The procedure stops either when the list is empty or when the end of the list is reached.

3.3.2 *Upwards*

Upwards Small QoS Started Servers First - USQoSS. Clients are sorted by non-decreasing order of q_i (and non-increasing order of r_i in case of tie). For each client i in the list we search for an appropriate server: we take the next server on the way up to the root (i.e., an inner node that is already equipped with a replica) which has enough remaining capacity to treat all the client's requests. Of course the QoS-constraints of the client have to be respected. If there is no server, we take the first inner node j that satisfies $\mathsf{W}_j \geq r_i$ within the QoS-range and we place a replica in j. If we still find no appropriate node, this heuristic has no feasible solution.

Upwards Small QoS Minimal Requests - USQoSM. This heuristic processes the clients in the same order as the previous one, but the choice of the appropriate server differs. Among the nodes in the QoS-range of client i, the node j with minimal ($\mathsf{W}_j - \mathsf{inreqQoS}_j$)-value is chosen as a server if it can satisfy r_i requests. Again it may happen that the heuristic cannot find a feasible solution, whenever no inner node can be found for a client.

Upwards Minimal Distance - UMD. This heuristic requires two steps. In the first step, so-called indispensable servers are chosen, i.e., inner nodes which have a client that must be treated by this very node. At the beginning, all servers that have a child client with $\mathsf{q} = 1$ will be chosen. This step guarantees that in each loop of the algorithm, we do not forget any client. The criterion for indispensable servers is the following: for each client check the number of nodes eligible as servers; if there is only one, this node is indispensable and chosen. The second step of UMD chooses the inner node with minimal ($\mathsf{W}_j - \mathsf{inreqQoS}_j$)-value as server (if $\mathsf{inreqQoS}_j > 0$). Note that this value can be negative. Then clients are associated to this server in order of distance, i.e., clients that are close to the server are chosen first, until the server capacity W_j is reached or no further client can be found.

3.3.3 *Multiple*

Multiple Small QoS Close Servers First - MSQoSC. The main idea of this heuristic is the same as for USQoSS, but with two differences. Searching for an appropriate server, we take the first inner node on the way up to the root which has some remaining capacity. Note that this makes the difference between *close* and *started* servers. If this capacity W_i is not sufficient

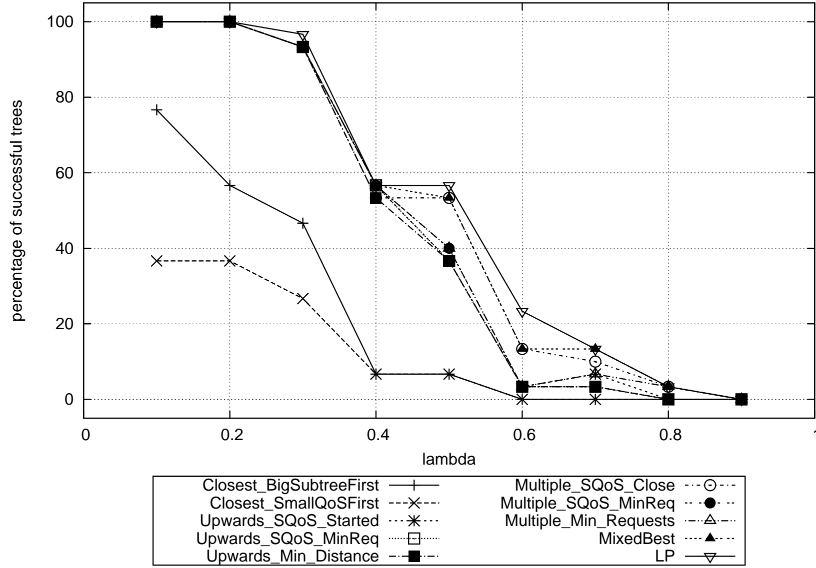


Figure 3.7: Success of small trees with tight QoS constraints, $\mathsf{q} \in \{1, 2\}$.

(client c has more requests, $\mathsf{W}_i < r_c$), we choose other inner nodes going upwards to the root until all requests of the client can be processed (this is possible owing to the multiple-server relaxation). If we cannot find enough inner nodes for a client, this heuristic will not return a feasible solution.

Multiple Small QoS Minimal Requests - MSQoSM. In this heuristic clients are treated in non-decreasing order of q_i, and the appropriate servers j are chosen by minimal $(\mathsf{W}_j - \mathsf{inreqQoS}_j)$-value until all requests of clients can be processed.

Multiple Minimal Requests - MMR. This heuristic is the counterpart of UMD for the *Multiple* policy and requires two steps. policy: servers Servers are added in the "indispensable" step, either when they are the only possible server for a client, or when the total capacity of all possible inner nodes for a client i is exactly r_i. The server chosen in the second step is also the inner node with minimal $(\mathsf{W}_j - \mathsf{inreqQoS}_j)$-value, but this time clients are associated in non-decreasing order of $\min(\mathsf{q}_i, d(i, r))$, where $d(i, r)$ is the number of hops between i and the root of the tree. Note that the last client that is associated to a server, might not be processed entirely by this server.

Mixed Best - MB. This heuristic unifies all previous ones. For each tree, we select the best cost returned by the other heuristics. Since each solution for *Closest* is also a solution for *Upwards*, which in turn is a valid solution for *Multiple*, this heuristic provides a solution for the *Multiple* policy.

3.4 Experimental Plan

In this section we evaluate the performance of our heuristics on tree platforms with varying parameters. Through these experiments we want to assess the different access policies, and the impact of QoS constraints on the performance of the heuristics. We obtain an optimal solution for each tree platform with the help of a mixed integer linear program (that we discussed in

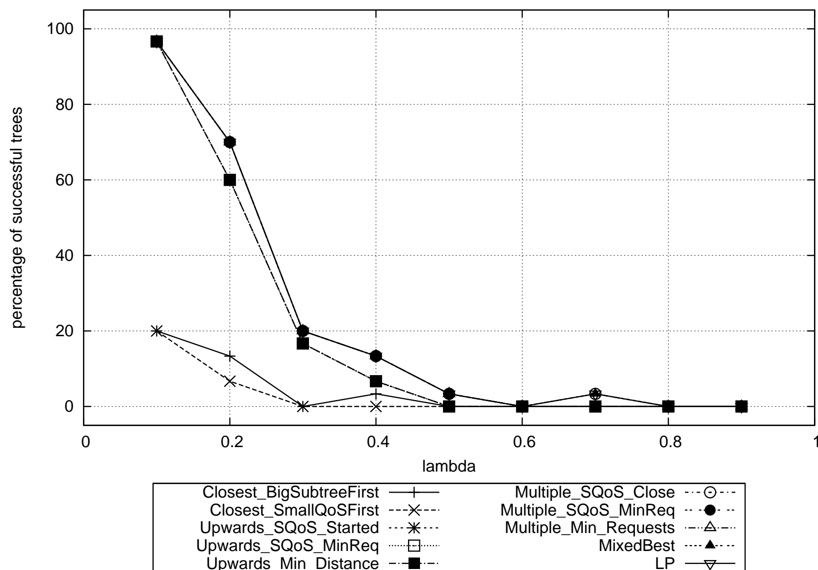


Figure 3.8: Success of big trees with tight QoS constraints, $\mathsf{q} \in \{1, 2\}$.

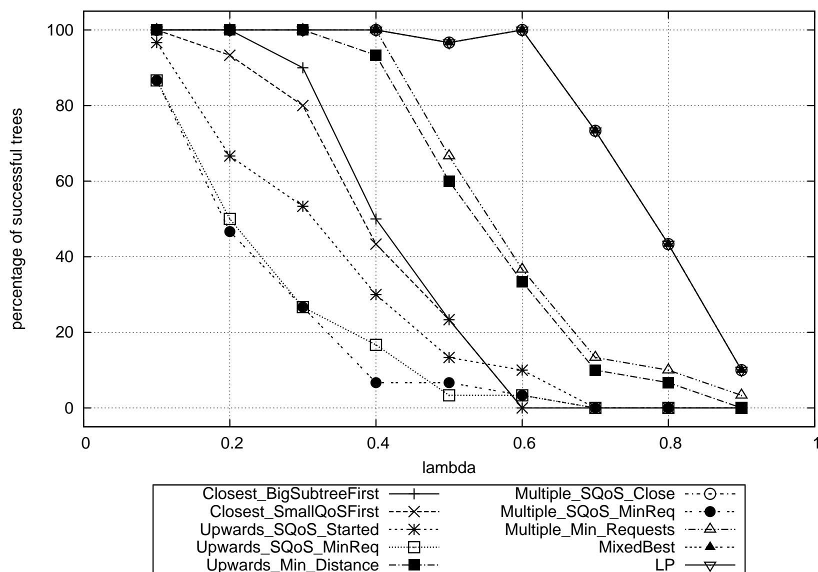


Figure 3.9: Success of small trees without QoS constraints, $\mathsf{q} = \mathsf{height} + 1$.

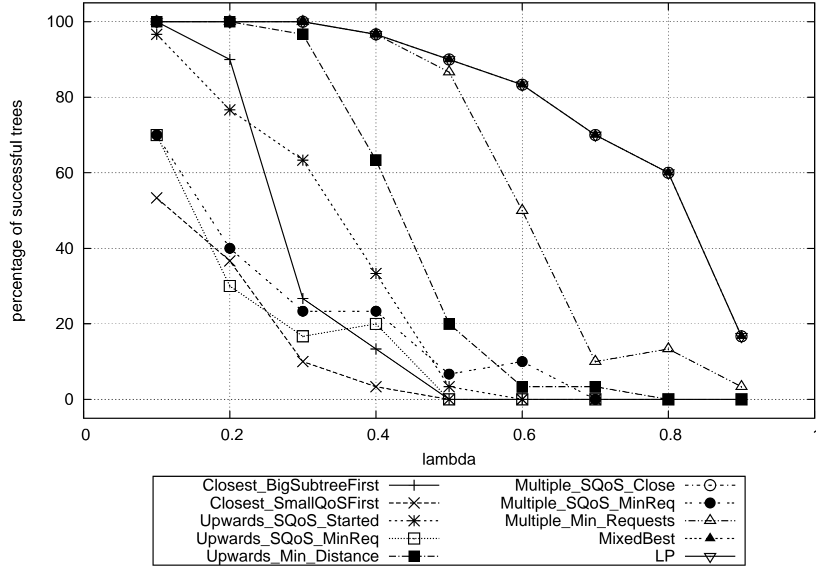


Figure 3.10: Success of big trees without QoS constraints $\mathsf{q} = \mathsf{height} + 1$.

Section 3.2). We can compute the latter optimal solution for problem sizes up to 400 nodes and clients, using GLPK [34].

Our experimental plan is similar to the one for experiments with mono-criteria heuristics in Section 2.4. Remember the parameter for the load in our tree networks $\lambda = \frac{\sum_{i \in \mathcal{C}} r_i}{\sum_{j \in \mathcal{N}} \mathsf{W}_j}$, where $\mathcal{C}$ is the set of clients in the tree and $\mathcal{N}$ the set of inner nodes. We tested our heuristics for $\lambda = 0.1, 0.2, ..., 0.9$, each on 30 randomly generated trees of two heights: in a first series, trees have a height between 4 and 7 (small trees). In the second series, tree heights vary between 16 and 21 (big trees). All trees have s nodes, where $15 \leq s \leq 400$. To assess the impact of QoS on the performance, we study the behavior (i) when QoS constraints are very tight ($\mathsf{q} \in \{1, 2\}$); (ii) when QoS constraints are more relaxed (the average value is set to half of the tree height height); and (iii) without any QoS constraint ($\mathsf{q} = \mathsf{height} + 1$).

We have computed the number of solutions for each λ and each heuristic. The number of solutions obtained by the linear program indicates which problems are solvable. Of course we cannot expect a result with our heuristics for intractable problems. To assess the performance of our heuristics, we have studied the relative performance of each heuristic compared to the optimal solution. This allows to compare the cost of the different heuristics, and thus to compare the different access policies. Recall: For each λ, the cost is computed on those trees for which the linear program has a solution. Let T_λ be the subset of trees with a LP solution. Then, the relative performance for the heuristic h is obtained by $\frac{1}{|T_\lambda|} \sum_{t \in T_\lambda} \frac{cost_{LP}(t)}{cost_h(t)}$, where $cost_{LP(t)}$ is the optimal solution cost returned by the linear program on tree t, and $cost_h(t)$ is the cost involved by the solution proposed by heuristic h. In order to be fair versus heuristics that have a higher success rate, we set $cost_h(t) = +\infty$, if the heuristic did not find any solution.

Before we comment on our results in detail, we want to point out, that CBS outperforms CSQoS in all experiments. That is why we will only consider CBS in the following, when speaking of the *Closest* policy.

Figures 3.7 and 3.9 show the percentage of success of each heuristic for small trees, while

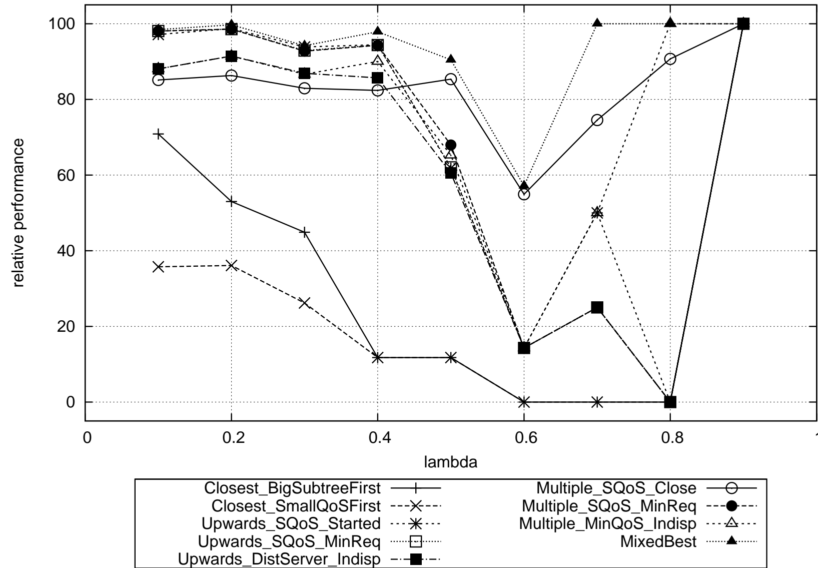


Figure 3.11: Relative performance of small trees with tight QoS constraints, $\mathsf{q} \in \{1, 2\}$.

the percentage of success for big trees is shown in Figures 3.8 and 3.10. A general overview of all figures shows that, as expected, the *Closest* policy has the poorest success rate for all its heuristics, whereas the *Multiple* heuristics almost always find a solution when the LP finds one. In fact, MB and MSQoSC always find a solution when the LP does with the exception of the configuration (small trees, $\lambda \geq 0.5$, $\mathsf{q} \in \{1, 2\}$). In this case the success rate is slightly inferior. The *Upwards* heuristic that finds the most solutions is UDS, followed by USQoSS. In the case of no QoS constraints (see Figures 3.9 and 3.10), the *Closest* heuristics outperform USQoSM and MSQoSM for small values of λ. In general MSQoSM finds fewer solutions than other *Multiple* heuristics.

Figures 3.11 to 3.14 give an overview[2] of our performance tests. The comparison between Figure 3.11 and 3.13 shows the impact of QoS on the performance. The impact of the tree sizes can be seen by comparing Figure 3.12 and 3.14. Globally, all the results show that QoS constraints do not modify the relative performance of the three policies: with or without QoS, *Multiple* is better than *Upwards*, which in turn is better than *Closest*, and their difference in performance is not sensitive to QoS tightness or to tree sizes. This is an enjoyable result, that could not be predicted a priori. The MB heuristic returns very good results, being relatively close to the optimal in most cases. The best heuristic to use depends on the tightness of QoS constraints. Thus, for *Multiple*, MSQoSM is the best choice for tight QoS constraints and small λ (Figure 3.11).

Altogether we conclude, when QoS is very restricting and λ small, that MSQoSM is the best choice. When QoS is less constrained, MMR is the best for λ up to 0.4. For big λ, MSQoSC is to prefer, since it never performs poorly in this case. In the case of less restricting QoS values, we choose MMR for λ up to 0.4 and then MSQoSC. Generally, when λ is high, MSQoSC never performs poorly. Concerning the *Upwards* policy, USQoSS behaves the best for tight QoS, in the other cases UMD achieves better results.

[2]The complete set of results can be found on the Web at
`http://graal.ens-lyon.fr/~vsonigo/code/replicaQoS/`

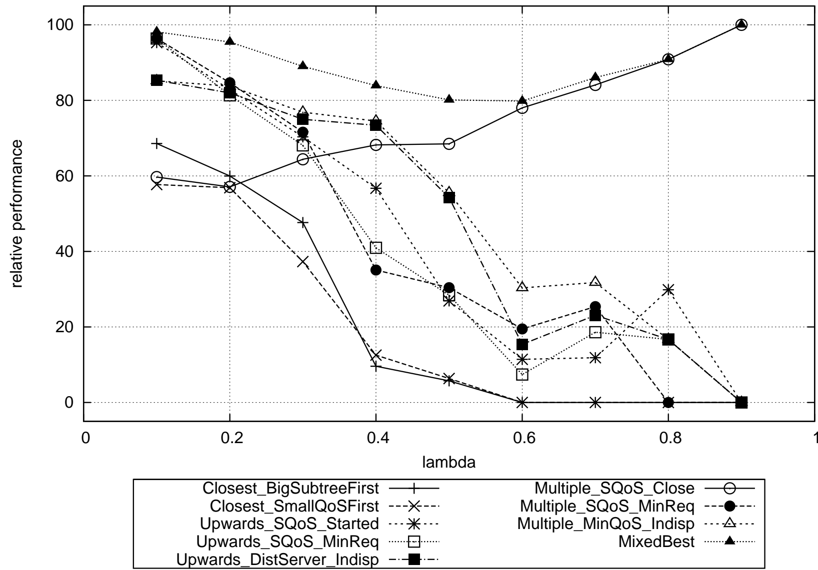


Figure 3.12: Relative performance of small trees with medium QoS constraints, $\mathsf{q} = \mathsf{height}/2$.

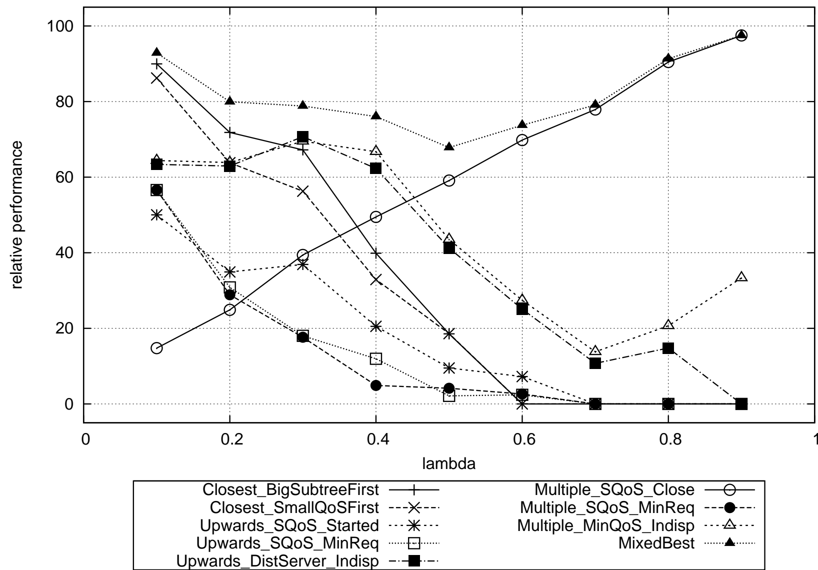


Figure 3.13: Relative performance of small trees without QoS constraints, $\mathsf{q} = \mathsf{height} + 1$.

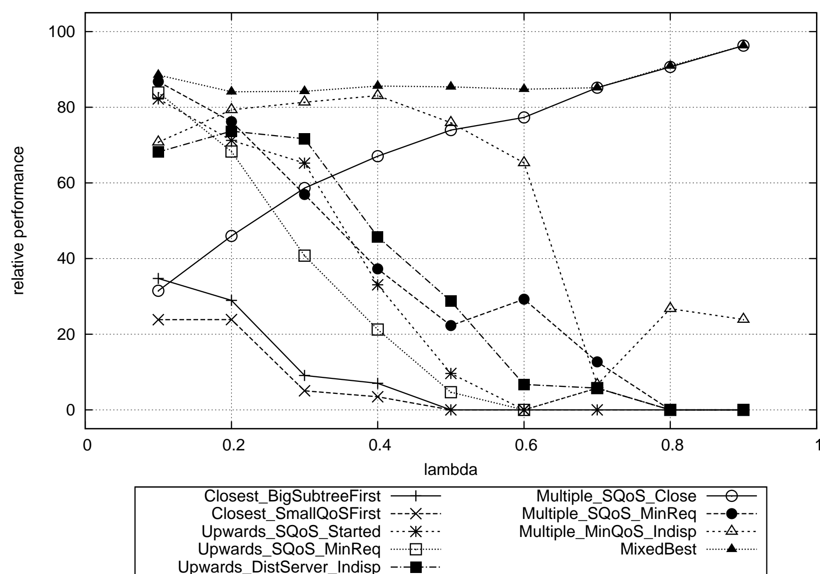


Figure 3.14: Relative performance of big trees with medium QoS constraints, $\mathsf{q} = \mathsf{height}/2$.

Part II

Pipeline Workflow Applications

Chapter 4

Problem Definition

This part deals with another mapping problem, namely the mapping of workflow applications onto heterogeneous platforms. Instead of mapping replicas onto a tree network we concentrate in this part on linear pipeline graph applications. As is well known, mapping applications onto parallel platforms is a difficult challenge. Several scheduling and load-balancing techniques have been developed for homogeneous architectures (see [63] for a survey) but the advent of heterogeneous clusters has rendered the mapping problem even more difficult. Typically, such clusters are composed of different-speed processors interconnected either by plain Ethernet (the low-end version) or by a high-speed switch (the high-end counterpart), and they constitute the experimental platform of choice in most academic or industry research departments. Moreover, in a distributed computing architecture, some processors may suddenly become unavailable, and we are facing the problem of failure [3, 6].

In this context of heterogeneous platforms, a structured programming approach rules out many of the problems which the low-level parallel application developer is usually confronted to, such as deadlocks or process starvation. Moreover, many real applications draw from a range of well-known solution paradigms, such as pipelined or farmed computations. High-level approaches based on algorithmic skeletons [23, 57] identify such patterns and seek to make it easy for an application developer to tailor such a paradigm to a specific problem. A library of skeletons is provided to the programmer, who can rely on these already coded patterns to express the communication scheme within its own application. Moreover, the use of a particular skeleton carries with it considerable information about implied scheduling dependencies, which we believe can help address the complex problem of mapping a distributed application onto a heterogeneous platform.

In this part, we consider application workflows that can be expressed as pipeline graphs. Typical applications include digital image processing, where images have to be processed in steady-state mode. A well known pipeline application of this type is for example JPEG encoding (see http://www.jpeg.org/). In such workflow applications, a series of data sets (tasks) enter the input stage and progress from stage to stage until the final result is computed. Each stage has its own communication and computation requirements: it reads an input file from the previous stage, processes the data and outputs a result to the next stage. For each data set, initial data is input to the first stage, and final results are output from the last stage. The pipeline workflow operates in synchronous mode: after some latency due to the initialization delay, a new task is completed every period. The period is defined as the longest cycle-time to operate a stage.

Each processor has a failure probability, which expresses the chance that the processor fails during execution. Key metrics for a given workflow are the throughput, the latency, and the

failure probability. The throughput measures the aggregate rate of processing of data, and it is the rate at which data sets can enter the system. Equivalently, the inverse of the throughput, defined as the period, is the time interval required between the beginning of the execution of two consecutive data sets. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. Note that it may well be the case that different data sets have different latencies (because they are mapped onto different processor sets), hence the latency is defined as the maximum response time over all data sets. Intuitively, we minimize the latency by assigning all stages to the fastest processor, but this may lead to an unreliable execution of the application. Minimizing the latency is antagonistic to minimizing the period as well it is antagonistic to minimizing the failure probability, and trade-offs should be found between these criteria. We focus on several bi-criteria approaches:

(i) minimizing the latency under period constraints, or the converse.
(ii) minimizing the failure probability under latency constraints, or the converse.

The problem of mapping pipeline skeletons onto parallel platforms has received some attention, and we survey related work in Section 4.3. We target heterogeneous clusters, and aim at deriving optimal mappings for the following bi-criteria objective functions: (i) mappings which minimize the period for a fixed maximum latency, or which minimize the latency for a fixed maximum period; (ii) mappings which minimize the failure probability for a fixed maximum latency, or which minimize the latency for a fixed maximum failure probability. We require the mapping to be interval-based, i.e., a processor is assigned an interval of consecutive stages.

Each pipeline stage can be seen as a sequential procedure which may perform disc accesses or write data in the memory for each task. This data may be reused from one task to another, and thus the rule of the game is always to process the tasks in a sequential order within a stage. Moreover, due to the possible local memory accesses, a given stage must be mapped onto a single processor: we cannot process half of the tasks on a processor and the remaining tasks on another without exchanging intra-stage information, which might be costly and difficult to implement. However, in order to improve reliability, we can replicate the computations for a given stage on several processors, i.e., a set of processors performs identical computations on every data set. Thus, in case of failure, we can take the result from a processor which is still working. In other words, a processor that is assigned a stage will execute the operations required by this stage (input, computation and output) for all the tasks fed into the pipeline. Note that replication of computations is only allowed in order to improve reliability.

The optimization problem can be stated informally as follows: which stage to assign to which (set of) processors?

This part is organized as follows: Chapter 4 introduces the framework (Section 4.1) and gives some motivating examples (Section 4.2). Related work is subject of Section 4.3. Chapter 5 is devoted to our complexity results. In Chapter 6 we present several polynomial time heuristics for the latency-period approach that are exhaustively tested in a case study.

4.1 Framework

4.1.1 Applicative Framework

The application is expressed as a pipeline graph of n stages $\mathcal{S}_k$, $1 \leq k \leq n$, as illustrated on Figure 4.1. Consecutive data sets are fed into the pipeline and processed from stage to stage,

until they exit the pipeline after the last stage. Each stage executes a task. More precisely, the k-th stage $\mathcal{S}_k$ receives an input from the previous stage, of size δ_{k-1}, performs a number of w_k computations, and outputs data of size δ_k to the next stage. This operation corresponds to the k-th task and is repeated periodically on each data set. The first stage $\mathcal{S}_1$ receives an input of size δ_0 from the outside world, while the last stage $\mathcal{S}_n$ returns the result, of size δ_n, to the outside world.

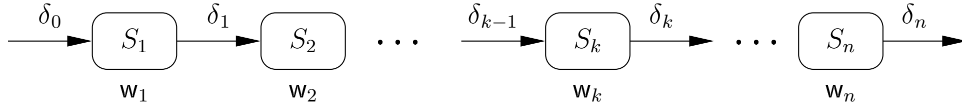


Figure 4.1: The application pipeline.

4.1.2 Target Platform

We target a platform (see Figure 4.2), with p processors P_u, $1 \leq u \leq p$, fully interconnected as a (virtual) clique.

We associate to each processor a failure probability $0 \leq \mathsf{fp}_u \leq 1$, $1 \leq u \leq p$, which is the probability that the processor breaks down during the execution of the application. A set of processors with identical failure probabilities is denoted *Failure Homogeneous* and otherwise *Failure Heterogeneous*. We consider a constant failure probability as we are dealing with workflows. These workflows are meant to run during a very long time, and therefore we address the question of whether the processor will break down or not at any time during execution. Indeed the maximum latency will be determined by the latency of the datasets which are processed after the failure.

There is a bidirectional link $\mathsf{link}_{u,v} : P_u \rightarrow P_v$ between any processor pair P_u and P_v, of bandwidth $\mathsf{b}_{u,v}$. Note that we do not need to have a physical link between any processor pair. Instead, we may have a switch, or even a path composed of several physical links, to interconnect P_u and P_v; in the latter case we would retain the bandwidth of the slowest link in the path for the value of $\mathsf{b}_{u,v}$.

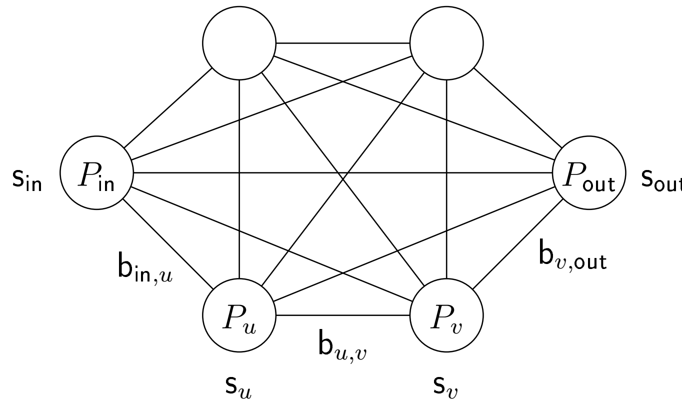


Figure 4.2: The target platform.

The speed of processor P_u is denoted as s_u, and it takes X/s_u time-units for P_u to execute X floating point operations. We also enforce a linear cost model for communications, hence it takes X/b time-units to send (resp. receive) a message of size X to (resp. from) P_v. Communications contention is taken care of by enforcing the *one-port* model [14]. In this model, a given processor can be involved in a single communication at any time-step, either a send or a receive. However, independent communications between distinct processor pairs can take place simultaneously. The one-port model seems to fit the performance of some current MPI implementations, which serialize asynchronous MPI sends as soon as message sizes exceed a few megabytes [59].

We consider three types of platforms:

- *Fully Homogeneous* platforms have identical processors ($\mathsf{s}_u = \mathsf{s}$ for $1 \leq u \leq p$) and interconnection links ($\mathsf{b}_{u,v} = \mathsf{b}$ for $1 \leq u, v \leq p$);
- *Communication Homogeneous* platforms, with identical links but different speed processors, introduce a first degree of heterogeneity;
- *Fully Heterogeneous* platforms constitute the most difficult instance, with different speed processors and different capacity links.

Finally, we assume that two special additional processors P_{in} and P_{out} are devoted to input/output data. Initially, the input data for each task resides on P_{in}, while all results must be returned to and stored in P_{out}.

4.1.3 Mapping Problem

The general mapping problem consists in assigning application stages to platform processors. For the sake of simplicity, we can assume that each stage $\mathcal{S}_k$ of the application pipeline is mapped onto a distinct processor (which is possible only if $n \leq p$). However, such one-to-one mappings may be unduly restrictive, and a natural extension is to search for interval mappings, i.e., allocation functions where each participating processor is assigned an interval of consecutive stages. Intuitively, assigning several consecutive tasks to the same processors will increase their computational load, but may well dramatically decrease communication requirements. In fact, the best interval mapping may turn out to be a one-to-one mapping, or instead may enroll only a very small number of fast computing processors interconnected by high-speed links.

Interval mappings constitute a natural and useful generalization of one-to-one mappings (not to speak of situations where $p < n$, where interval mappings are mandatory), and such mappings have been studied by Subhlock et al. [66, 67].

Formally, we search for a partition of $[1..n]$ into $m \leq p$ intervals $I_j = [d_j, e_j]$ such that $d_j \leq e_j$ for $1 \leq j \leq m$, $d_1 = 1$, $d_{j+1} = e_j + 1$ for $1 \leq j \leq m - 1$ and $e_p = n$.

The function $\mathsf{alloc}(j)$ returns the indices of the processors on which interval I_j is mapped. There are $k_j = |\mathsf{alloc}(j)|$ processors executing I_j, and obviously $k_j \geq 1$. Increasing k_j increases the reliability of the execution of interval I_j. We assume that $\mathsf{alloc}(0) = \{\mathsf{in}\}$ and $\mathsf{alloc}(p+1) = \{\mathsf{out}\}$, where P_{in} is a special processor holding the initial data, and P_{out} is receiving the results.

Period

Dealing with *Fully Homogeneous* and *Communication Homogeneous* platforms, the period can be expressed as

$$\mathcal{P} = \max_{1 \leq j \leq m} \left\{ \frac{\delta_{d_j - 1}}{\mathsf{b}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{\mathsf{s}_{\mathsf{alloc}(j)}} + \frac{\delta_{e_j}}{\mathsf{b}} \right\} \tag{4.1}$$

On *Fully Heterogeneous* platforms we have to take into account the heterogeneous communication links. This leads to

$$\mathcal{P} = \max_{1 \le j \le m} \left\{ \frac{\delta_{d_j - 1}}{\mathsf{b}_{u,v}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{\mathsf{s}_{\mathsf{alloc}(j)}} + \frac{\delta_{e_j}}{\mathsf{b}_{u,v}} \right\}$$

Note that we do not take failure probabilities into account, when we deal with period optimization. Hence we have no replication of intervals here and each stage is only mapped onto a single processor.

Reliability

The failure probability can be computed given the number m of intervals and the set of processors assigned to each interval:

$$\mathcal{FP} = 1 - \prod_{1 \le j \le m} (1 - \prod_{u \in \mathsf{alloc}(j)} \mathsf{fp}_u). \tag{4.2}$$

Latency

Dealing with *Fully Homogeneous* and *Communication Homogeneous* platforms, the latency is obtained as

$$\mathcal{L} = \sum_{1 \le j \le m} \left\{ k_j \times \frac{\delta_{d_j - 1}}{\mathsf{b}} + \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{\min_{u \in \mathsf{alloc}(j)} (\mathsf{s}_u)} \right\} + \frac{\delta_n}{\mathsf{b}}. \tag{4.3}$$

In equation (4.3), we consider the longest path required to compute a given data set. The worst case is when the first processors involved in the replication fail during execution. A communication to interval j must then be paid k_j times since these are serialized (one-port model). For computations, we consider the total computation time required by the slowest processor assigned to the interval. For the final output, only one communication is required, hence the δ_n/b. Note that in order to achieve this latency, we need a standard consensus protocol to determine which of the surviving processors performs the outgoing communications [70].

A similar mechanism is used for *Fully Heterogeneous* platforms:

$$\begin{aligned} \mathcal{L} = & \sum_{u \in \mathsf{alloc}(1)} \frac{\delta_0}{\mathsf{b}_{\mathsf{in},u}} + \\ & \sum_{1 \le j \le m} \max_{u \in \mathsf{alloc}(j)} \left\{ \frac{\sum_{i=d_j}^{e_j} \mathsf{w}_i}{\mathsf{s}_u} + \sum_{v \in \mathsf{alloc}(j+1)} \frac{\delta_{e_j}}{\mathsf{b}_{u,v}} \right\} \end{aligned} \tag{4.4}$$

Remark that we have $k_j = 1$, $1 \le j \le m$, when we abstract from failures, in particular in combination with period optimization.

Optimization problems

The optimization problem is to determine the best mapping, over all possible partitions into intervals, and over all processor assignments. The objective can be to minimize either the period, the latency or the failure probability, or a combination. We deal with the following combinations:

- Given a threshold period, what is the minimum latency that can be achieved? And the counterpart: Given a threshold latency, what is the minimum period that can be achieved?
- Given a threshold latency, what is the minimum failure probability that can be achieved? Similarly, given a threshold failure probability, what is the minimum latency that can be achieved?

4.2 Motivating Examples

Before presenting complexity results in Chapter 5, we want to make the reader more sensitive to the difficulty of the problem via some motivating examples.

We start with the mono-criterion interval mapping problem of minimizing the latency. For *Fully Homogeneous* and *Communication Homogeneous* platforms the optimal latency is achieved by assigning the whole pipeline to the fastest processor. This is due to the fact that mapping the whole pipeline onto one single processor minimizes the communication cost since all communication links have the same characteristics. Choosing the fastest processor on *Communication Homogeneous* platforms ensures the shortest processing time.

However, this line of reasoning does not hold anymore when communications become heterogeneous. Let us consider for instance the mapping of the pipeline of Figure 4.3 on the *Fully Heterogeneous* platform of Figure 4.4. The pipeline consists of two stages, both needing the same amount of computation ($\mathsf{w} = 2$), and the same amount of communications ($\delta = 100$). In this example, a mapping which minimizes the latency must map each stage on a different processor, thus splitting the stages into two intervals. In fact, if we map the whole pipeline on a single processor, we achieve a latency of $100/100 + (2 + 2)/1 + 100/1 = 105$, either if we choose P_1 or P_2 as target processor. Splitting the pipeline and hence mapping the first stage on P_1 and the second stage on P_2 requires to pay the communication between P_1 and P_2 but drastically decreases the latency: $100/100 + 2/1 + 100/100 + 2/1 + 100/100 = 1 + 2 + 1 + 2 + 1 = 7$.

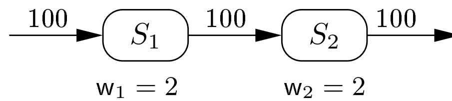


Figure 4.3: Example optimal with 2 intervals.

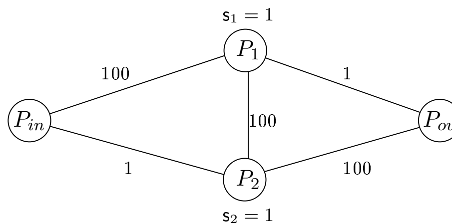


Figure 4.4: The pipeline has to be split into intervals to achieve an optimal latency on this platform.

Unfortunately these intuitions cannot be generalized when tackling bi-criteria optimization, where latency should be minimized respecting a certain failure threshold or the converse. We

will prove in Lemma 5.1 that minimizing the failure probability under a fixed latency threshold on *Fully Homogeneous* and *Communication Homogeneous-Failure Homogeneous* platforms still can be done by keeping a single interval.

However, if we consider *Communication Homogeneous-Failure Heterogeneous*, we can find examples in which this property is not true. Consider for instance the pipeline of Figure 4.5. The target platform consists of one processor of speed 1 and failure probability 0.1, it is a slow but reliable processor. On the other hand we have 10 fast and unreliable processors, of speed 100 and failure probability 0.8. All communication links have a bandwidth $\mathsf{b} = 1$. If the latency threshold is fixed to 22, the slow processor cannot be used in the replication scheme. Also, if we use three fast processors, the latency is $3 * 10 + 101/100 > 22$. Thus the best one-interval solution reaches a failure probability of $(1 - (1 - 0.8^2)) = 0.64$, which is very high. We can do much better by using the slow processor on the slow stage, and then replicate ten times the second stage on the fast processors, achieving a latency of $10 + 1/1 + 10 * 1 + 100/100 = 22$ and a failure probability of $1 - (1 - 0.1).(1 - 0.8^{10}) < 0.2$. Thus the optimal solution does not consist of a single interval in this case.

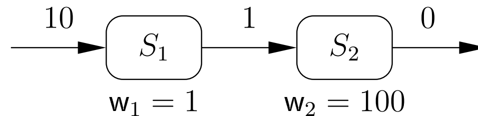


Figure 4.5: Example optimal with 2 intervals.

4.3 Related Work

This work is an extension of the work of Subhlock and Vondran [66, 67] for pipeline applications on homogeneous platforms. We have extended their complexity results to heterogeneous platforms.

Several papers consider the problem of mapping communicating tasks onto heterogeneous platforms, but for a different applicative framework. In [69], Taura and Chien consider applications composed of several copies of the same task graph, expressed as a DAG (directed acyclic graph). These copies are to be executed in pipeline fashion. Taura and Chien also restrict to mapping all instances of a given task type (which corresponds to a stage in our framework) onto the same processor. Their problem is shown NP-complete, and they provide an iterative heuristic to determine a good mapping. At each step, the heuristic refines the current clustering of the DAG. Beaumont et al. [9] consider the same problem as Taura and Chien, i.e., with a general DAG, but they allow a given task type to be mapped onto several processors, each executing a fraction of the total number of tasks. The problem remains NP-complete, but becomes polynomial for special classes of DAGs, such as series-parallel graphs. For such graphs, it is possible to determine the optimal mapping owing to an approach based upon a linear programming formulation. Due to complex mapping rules, the approach of [9] can only achieve optimal throughput through very long periods: hence the simplicity and regularity of the schedule are lost, while the latency is severely increased. On the contrary, the simpler mapping rules used in this work allow for better period/latency trade-offs.

Another important series of papers comes from the DataCutter project [27]. One goal of this project is to schedule multiple data analysis operations onto clusters and grids, decide where to place and/or replicate various components [12, 13, 64]. A typical application is a chain

of consecutive filtering operations, to be executed on a very large data set. The task graphs targeted by DataCutter are more general than linear pipelines or forks, but still more regular than arbitrary DAGs, which makes it possible to design efficient heuristics to solve the previous placement and replication optimization problems.

A recent paper [72] targets generalized workflows structured as arbitrary DAGs, and considers an instance of the bi-criteria optimization problem where the latency is optimized under a fixed throughput constraint. Only completely homogeneous platforms are considered in [72]. It would be very interesting (but also very challenging) to extend the heuristics of [72] to heterogeneous frameworks, and to compare them with the heuristics that we specifically designed for pipeline workflows.

In the context of embedded systems, energy consumption is another important objective to minimize. Three-criteria optimization (energy, latency and throughput) is discussed in [77].

The former related works dealt mostly with generalized applications which assume synthetic workload and application pipelines. In our case-study (Chapter 6) we apply the bi-criteria mapping problem to a concrete application pipeline in digital image encoding, the JPEG encoder. In this domain, parallelization strategies have been considered beforehand, but the pipelined nature of the applications has not been fully exploited yet.

Many authors, started from the blockwise independent processing of the JPEG encoder in order to apply simple data parallelism for efficient parallelization. This fine-grain parallelization opportunity is for instance exploited in [29, 62]. In addition, parallelization of almost all stages, from color space conversion, over DCT to the Huffman encoding has been addressed [4, 44]. Recently, with respect to the JPEG2000 codec, efficient parallelization of wavelet coding has been introduced [49]. All these works target the best speed-up with respect to different architectures and possible varying load situations. Optimizing the period and the latency is an important issue when encoding a pipeline of multiple images, as for instance for Motion JPEG (M-JPEG). To meet these issues, one has to solve in addition to the above mentioned work a bi-criteria optimization problem, i.e., optimize the latency, as well as the period. The application of coarse grain parallelism seems to be a promising solution. We propose to use an interval-based mapping strategy allowing multiple stages to be mapped to one processor which allows meeting the most flexible the domain constraints (even for very large pictures). Several pipelined versions of the JPEG encoding have been considered. They rely mainly on pixel or block-wise parallelization [30, 53]. For instance, Ferretti et al. [30] uses three pipelines to carry out concurrently the encoding on independent pixels extracted from the serial stream of incoming data. The pixel and block-based approach is however useful for small pictures only. Recently, Sheel et al. [61] consider a pipeline architecture where each stage presents a step in the JPEG encoding. The targeted architecture consists of Xtensa LX processors which run subprograms of the JPEG encoder program. Each program accepts data via the queues of the processor, performs the necessary computation, and finally pushes it to the output queue into the next stage of the pipeline. The basic assumptions are similar to our work, however no optimization problem is considered and only runtime (latency) measurements are available. The schedule is static and set according to basic assumptions about the image processing, e.g., that the DCT is the most complex operation in runtime.

Chapter 5

Complexity Results

Dealing with mono-criterion optimization, we state that the difficulty of optimization highly depends on the optimization parameter. Minimizing the latency or the failure probability is trivial, while minimizing the period is NP-hard as soon as heterogeneity occurs. Quite interestingly, this last result is a consequence of the fact that the natural extension of the chains-to-chains problem [55] to different-speed processors is NP-hard.

Minimizing bi-criteria problems is a lot harder. While we can still provide optimal polynomial-time algorithms on homogeneous platforms, on heterogeneous platforms all problems are NP-complete.

In the following we first concentrate on the mapping problems with mono-criteria optimization and then pass to bi-criteria optimization.

5.1 Mono-criteria Problems

Table 5.1 gives an overview of the complexity of the different instances of the mono-criteria optimization problems.

5.1.1 Failure Probability

The problem of minimizing the failure probability can easily be solved for all types of platforms.

Theorem 5.1. *Minimizing the failure probability can be done in polynomial time.*

Proof. This can be seen easily from the formula computing the global failure probability: the minimum is reached by replicating the whole pipeline as a single interval on all processors. This is true for all platform types. ∎

Objective	*Fully Homogeneous*	*Communication Homogeneous*	*Fully Heterogeneous*
P	polynomial [66]	NP-hard[10]	NP-hard
L	polynomial	polynomial	NP-hard [11]
FP	polynomial	polynomial	polynomial

Table 5.1: Complexity results for the mono-criteria optimization problems.

5.1.2 Latency

The problem of minimizing the latency is trivially of polynomial time complexity for *Fully Homogeneous* and *Communication Homogeneous* platforms. However the problem becomes harder for *Fully Heterogeneous* platforms because of the first and last communications, which should be mapped on fast communicating links to optimize the latency. Notice that replication can only decrease latency so we do not consider any replication in this mono-criterion problem. However, we need to find the best partition of stages into intervals.

Theorem 5.2. *Minimizing the latency can be done in polynomial time on Communication Homogeneous platforms.*

Proof. The latency is optimized when we suppress all communications. Also, replication is increasing latency by adding extra communications. The minimum latency can be achieved by mapping the whole interval onto the fastest processor j, resulting in the latency $\left(\sum_{i=1}^{n} \mathsf{w}_i\right)/\mathsf{s}_j$. If a slower processor is involved in the mapping, the latency increases, following equation (4.3), since part of the computations will take longer, and communications may occur. ∎

In [11], Benoit et al. prove that minimizing the latency on *Fully Heterogeneous* platforms is NP-hard. We provide two further results for the latency minimization problem for linear pipeline graphs. First, if we relax the interval constraint, i.e., a set of non-consecutive stages can be assigned to a same processor, then the problem becomes polynomial. We call such mappings *general mappings*. Second, considering one-to-one mappings, where each stage is mapped onto a different processor, the problem is NP-hard.

Theorem 5.3. *Minimizing the latency is polynomial on Fully Heterogeneous platforms for general mappings.*

Proof. We consider *Fully Heterogeneous* platforms and we want to minimize the latency.

Let us consider a directed graph with $n.m+2$ vertices, and $(n-1)m^2+2m$ edges, as illustrated in Figure 5.1. $V_{i,u}$ corresponds to the mapping of stage $\mathcal{S}_i$ onto processor P_u. $V_{0,\mathsf{in}}$ and $V_{(n+1),\mathsf{out}}$ represent the initial and final processors, and data must flow from $V_{0,\mathsf{in}}$ to $V_{(n+1),\mathsf{out}}$. Edges represent the flow of data from one stage to another, thus we have m^2 edges for $i = 0..n$, connecting vertex $V_{i,u}$ to $V_{i+1,v}$ for $u, v = 1..m$ (except for the first and last stages where there are only m edges).

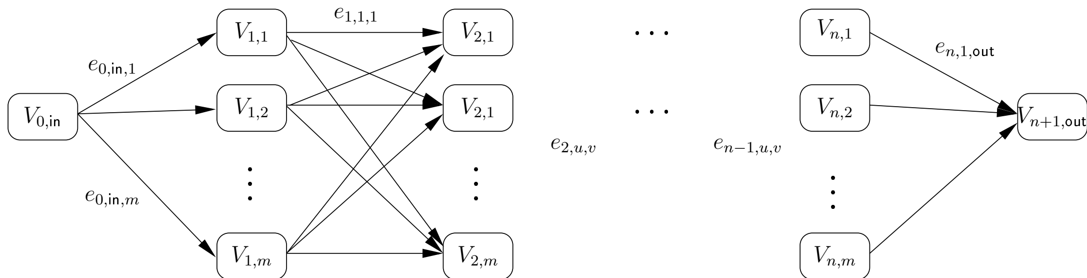


Figure 5.1: Minimizing the latency.

Thus, a general mapping can be represented by a path from $V_{0,\mathsf{in}}$ to $V_{(n+1),\mathsf{out}}$: if $V_{i,u}$ is in the path then stage $\mathcal{S}_i$ is mapped onto P_u. Notice that a path can create intervals of non-consecutive stages, thus this mapping is not interval-based.

We assign weights to the edges to ensure that the weight of a path is the latency of the corresponding mapping. Computation cost of stage $\mathcal{S}_i$ on P_u is added on the m edges exiting $V_{i,u}$, and thus $e_{i,u,v} = \frac{\mathsf{w}_i}{\mathsf{s}_u}$. Communication costs are added on all edges: $e_{i,u,v} += \frac{\delta_i}{\mathsf{b}_{u,v}}$ if $P_u \neq P_v$. Edges $e_{i,u,u}$ correspond to intra-interval communications, and thus there is no communication cost to pay.

The mapping which realizes the minimum latency can be obtained by finding a shortest path in this graph going from $V_{0,\mathsf{in}}$ to $V_{(n+1),\mathsf{out}}$. The graph has polynomial size and the shortest path can be computed in polynomial time [25], thus we have the result in polynomial time, which concludes the proof.

■

Now assuming the simplest mappings, where each stage $\mathcal{S}_k$ of the application pipeline is mapped onto a distinct processor (which is possible only if $n \leq p$), we obtain the following result:

Theorem 5.4. *Minimizing the latency is NP-hard on Fully Heterogeneous platforms for one-to-one mappings.*

Proof. The problem clearly belongs to NP. We use a reduction from the Traveling Salesman Problem (TSP), which is NP-complete [32]. Consider an arbitrary instance $\mathcal{I}_1$ of TSP, i.e., a complete graph $G = (V, E, c)$, where $c(e)$ is the cost of edge e, a source vertex $s \in V$, a tail vertex $t \in V$, and a bound K: is there an Hamiltonian path in G from s to t whose cost is not greater than K?

We build the following instance $\mathcal{I}_2$ of the one-to-one latency minimization problem: we consider an application with $n = |V|$ identical stages. All application costs are unit costs: $w_i = \delta_i$ for all i. For the platform, in addition to P_{in} and P_{out} we use $m = n = |V|$ identical processors of unit speed: $s_i = 1$ for all i. We simply write i for the processor P_i that corresponds to vertex $v_i \in V$.

We only play with the link bandwidths: we interconnect P_{in} and s, P_{out} and t with links of bandwidth 1. We interconnect i and j with a link of bandwidth $\frac{1}{c(e_{i,j})}$. All the other links are very slow (say their bandwidth is smaller than $\frac{1}{K+n+3}$). We ask whether we can achieve a latency $\mathcal{L} \leq K'$, where $K' = K + n + 2$. Clearly, the size of $\mathcal{I}_2$ is linear in the size of $\mathcal{I}_1$.

Because we have as many processors as stages, any solution to $\mathcal{I}_2$ will use all processors. We need to map the first stage on s and the last one on t, otherwise the input/output cost already exceeds K'. We spend 2 time-units for input/output, and n time-units for computing (one unit per stage/processor). There remain exactly K time-units for inter-processor communications, i.e., for the total cost of the Hamiltonian path that goes from s to t. We cannot use any slow link either. Hence we have a solution for $\mathcal{I}_2$ if and only if we have one for $\mathcal{I}_1$. ■

5.1.3 Period

Minimizing the period on *Fully Homogeneous* platforms can be done in polynomial time [66]. As we learned before, the failure probability can be minimized easily for all types of platforms and minimizing the latency can be done in polynomial time for *Communication Homogeneous* platforms. However, it is not so easy to minimize the period on *Communication Homogeneous* platforms, and in [10] Benoit and Robert prove that this problem is NP-hard.

Objective	Failure	*Fully Homogeneous*	*Communication Homogeneous*	*Fully Heterogeneous*
P & L	/	polynomial [67]	NP-hard	NP-hard
FP & L	hom.	polynomial	polynomial	NP-hard
FP & L	het.	polynomial	open	NP-hard

Table 5.2: Complexity results for the different instances of bi-criteria optimization

5.2 Bi-criteria Optimization

We give an overview of the complexity results for bi-criteria mapping problems in Table 5.2. As already pointed out, we either consider period-latency optimization problems or the combination of latency and failure probability.

5.2.1 Period and Latency

In [67], Subhlock et al. prove polynomiality of the bi-criteria mapping with period and latency constraints on *Fully Homogeneous* platforms. Since the period minimization problem is NP-hard for *Communication Homogeneous* platforms, all bi-criteria problems on *Communication Homogeneous* or *Fully Heterogeneous* platforms are NP-hard.

5.2.2 Latency and Failure Probability

Preliminary Lemma

We start with a preliminary lemma which proves that there is an optimal solution of both bi-criteria problems for latency and failure probability consisting of a single interval for *Fully Homogeneous* platforms, and for *Communication Homogeneous* platforms with identical failure probabilities.

Lemma 5.1. *On Fully Homogeneous and Communication Homogeneous-Failure Homogeneous platforms, there is a mapping of the pipeline as a single interval which minimizes the failure probability under a fixed latency threshold, and there is a mapping of the pipeline as a single interval which minimizes the latency under a fixed failure probability threshold.*

Proof. If the stages are split into p intervals, the failure probability is expressed as

$$1 - \prod_{1 \leq j \leq p} (1 - \prod_{u \in \mathsf{alloc}(j)} \mathsf{fp}_u).$$

Let us start with the *Fully Homogeneous* case, and with *Failure Heterogeneous* for a most general setting. We can transform the solution into a new one using a single interval, which improves both latency and failure probability. Let k_0 be the number of times that the first interval is replicated in the original solution. Then a solution which replicates the whole interval on the k_0 most reliable processors realizes: (i) a latency which is smaller since we remove the communications between intervals; (ii) a smaller failure probability since for the new solution $(1 - \prod_{u \in \mathsf{alloc}(1)} \mathsf{fp}_u)$ is greater than the same expression in the original solution (the most reliable processors are used in the new one), and moreover the old solution even decreases this value by multiplying it by other terms smaller than 1. Thus the new solution is better for both criteria.

In the case with *Communication Homogeneous* and *Failure Homogeneous*, we use a similar reasoning to transform the solution. We select the interval with the fewest number of processors,

denoted k. In the failure probability expression, there is a term in $(1-\mathsf{fp}^k)$, and thus the global failure probability is greater than $1-(1-\mathsf{fp}^k)$ which is obtained by replicating the whole interval onto k processors. Since we do not want to increase the latency, we use the fastest k processors, and it is easy to check that this scheme cannot increase latency ($k \leq k_0$ and the slowest processor is not slower than the slowest processor of any intervals of the initial solution). Thus the new solution is better for both criteria, which ends the proof.

We point out that Lemma 5.1 cannot be extended to *Communication Homogeneous* and *Failure Heterogeneous*: instead, we can build counter examples in which this property is not true, as illustrated in Section 4.2.

■

Fully Homogeneous Platforms

For *Fully Homogeneous* platforms, we consider that all failure probabilities are identical, since the platform is made of identical processors. However, results can easily be extended for different failure probabilities. We have seen in Lemma 5.1 that the optimal solution for a bi-criteria mapping on such platforms always consists in mapping the whole pipeline as a single interval. Otherwise, both latency and failure probability would be increased.

Theorem 5.5. *On Fully Homogeneous platforms, the solution to the bi-criteria problem of minimizing latency and failure probability can be found in polynomial time using Algorithm 5 or Algorithm 6.*

Informally, the algorithms find the maximum number of processors k that can be used in the replication set, and the whole interval is mapped on a set of k identical processors. With different failure probabilities, the more reliable processors are used.

Algorithm 5: *Fully Homogeneous* platforms: Minimizing $\mathcal{FP}$ for a fixed $\mathcal{L}$

---

begin

Find k maximum, such that

$$k \times \frac{\delta_0}{\mathsf{b}} + \frac{\sum_{1\leq j\leq n} \mathsf{w}_j}{\mathsf{s}} + \frac{\delta_n}{\mathsf{b}} \leq \mathcal{L}$$

Replicate the whole pipeline as a single interval onto the k (most reliable) processors;

end

Algorithm 6: *Fully Homogeneous* platforms: Minimizing $\mathcal{L}$ for a fixed $\mathcal{FP}$

---

begin

Find k minimum, such that

$$1-(1-\mathsf{fp}^k) \leq \mathcal{FP}$$

Replicate the whole pipeline as a single interval onto the k (most reliable) processors;

end

Proof. The proof of this theorem is based on Lemma 5.1. We prove it in the general setting of heterogeneous failure probabilities. An optimal solution can be obtained by mapping the pipeline

as a single interval, thus we need to decide the set of processors alloc used for replication. $|\mathsf{alloc}|$ is the number of processors used.

The first problem can be formally expressed as follows:

$$\begin{array}{l}\textsc{Minimize } 1-(1-\prod_{u\in\mathsf{alloc}}\mathsf{fp}_u),\\ \text{UNDER THE CONSTRAINT}\end{array} \tag{5.1}$$

$$|\mathsf{alloc}|\frac{\delta_0}{\mathsf{b}}+\frac{\sum_{1\leq i\leq n}\mathsf{w}_i}{\mathsf{s}}+\frac{\delta_n}{\mathsf{b}}\leq\mathcal{L}$$

This leads to minimize $\prod_{u\in\mathsf{alloc}}\mathsf{fp}_u$, and the constraint on the latency determines the maximum number k of processors which can be used:

$$k=\left\lfloor\frac{\mathsf{b}}{\delta_0}\left(\mathcal{L}-\frac{\delta_n}{\mathsf{b}}-\frac{\sum_{1\leq i\leq n}\mathsf{w}_i}{\mathsf{s}}\right)\right\rfloor$$

In order to minimize $\prod_{u\in\mathsf{alloc}}\mathsf{fp}_u$, we need to use as many processors as possible since $\mathsf{fp}_u\leq 1$ for $1\leq u\leq m$.

If one of the most reliable processors is not used, we can exchange it with a less reliable one, and thus increase the value of the product, so the formula is minimized when using the k most reliable processors, which is represented in Algorithm 5.

The second problem is expressed below:

$$\begin{array}{l}\textsc{Minimize } |\mathsf{alloc}|\frac{\delta_0}{\mathsf{b}}+\frac{\sum_{1\leq i\leq n}\mathsf{w}_i}{\mathsf{s}}+\frac{\delta_n}{\mathsf{b}},\\ \text{UNDER THE CONSTRAINT}\end{array} \tag{5.2}$$

$$1-(1-\prod_{u\in\mathsf{alloc}}\mathsf{fp}_u)\leq\mathcal{FP}$$

Latency increases when $|\mathsf{alloc}|$ is large, thus we need to find the smallest number of processors which satisfies constraint (4). As before, if one of the most reliable processors is not used, we can exchange it and improve the reliability without increasing the latency, which might lead to add fewer processors to the replication set for an identical reliability. Algorithm 6 thus returns the optimal solution.

■

Remark Both algorithms (5 and 6) are optimal as well in the case of heterogeneous failure probabilities. We add the most reliable processors to the replication scheme (thus increasing latency and decreasing the failure probability) while $\mathcal{L}$ or $\mathcal{FP}$ are not reached.

Communication Homogeneous Platforms

For *Communication Homogeneous* platforms, we first consider the simpler case where all failure probabilities are identical, denoted by *Failure Homogeneous*. In this case, the optimal bi-criteria solution still consists of the mapping of the pipeline as a single interval.

Theorem 5.6. *On Communication Homogeneous platforms with Failure Homogeneous, the solution to the bi-criteria problem of minimizing latency and failure probability can be found in polynomial time using Algorithm 7 or 8.*

Algorithm 7: *Communication Homogeneous* platforms - *Failure Homogeneous*: Minimizing $\mathcal{FP}$ for a fixed $\mathcal{L}$

begin
Order processors in non-increasing order of s_j;
Find k maximum, such that

$$k \times \frac{\delta_0}{\mathsf{b}} + \frac{\sum_{1 \le j \le n} \mathsf{w}_j}{\mathsf{s}_k} + \frac{\delta_n}{\mathsf{b}} \le \mathcal{L}$$

Replicate the whole pipeline as a single interval onto the fastest k processors;
// Note that at any time s_k is the speed of
// the slowest processor used
// in the replication scheme.
end

Algorithm 8: *Communication Homogeneous* platforms - *Failure Homogeneous*: Minimizing $\mathcal{L}$ for a fixed $\mathcal{FP}$

begin
Find k minimum, such that

$$1 - (1 - \mathsf{fp}^k) \le \mathcal{FP}$$

Replicate the whole pipeline as a single interval onto the fastest k processors;
end

Informally, we add the fastest processors to the replication set while the latency is not exceeded (or until $\mathcal{FP}$ is reached), thus reducing the failure probability and increasing the latency.

Proof. In this particular setting, Lemma 5.1 still applies, so we restrict to mappings as a single interval, and search for the optimal set of processors alloc which should be used.

The first problem is expressed as:

$$\begin{array}{l} \text{MINIMIZE } 1 - (1 - \mathsf{fp}^{|\mathsf{alloc}|}), \\ \text{UNDER THE CONSTRAINT} \end{array} \tag{5.3}$$

$$|\mathsf{alloc}|\frac{\delta_0}{\mathsf{b}} + \frac{\sum_{1 \le i \le n} \mathsf{w}_i}{\min_{u \in \mathsf{alloc}} \mathsf{s}_u} + \frac{\delta_n}{\mathsf{b}} \le \mathcal{L}$$

The failure probability is smaller when $|\mathsf{alloc}|$ is large, thus we need to add as many processors as we can while satisfying the constraint. The latency increases when adding more processors, and it depends of the speed of the slowest processors. Thus, if the $|\mathsf{alloc}|$ fastest processors are not used, we can exchange a fastest processor with a used one without increasing latency. Algorithm 7 thus returns an optimal mapping.

The other problem is similar, with the following expression:

$$\begin{array}{l} \text{MINIMIZE } |\mathsf{alloc}|\frac{\delta_0}{\mathsf{b}} + \frac{\sum_{1 \le i \le n} \mathsf{w}_i}{\min_{u \in \mathsf{alloc}} \mathsf{s}_u} + \frac{\delta_n}{\mathsf{b}}, \\ \text{UNDER THE CONSTRAINT} \end{array} \tag{5.4}$$

$$1 - (1 - \mathsf{fp}^{|\mathsf{alloc}|}) \leq \mathcal{FP}$$

We can thus find the smallest number of processors that should be used in order to satisfy $\mathcal{FP}$, and then use the fastest processors to optimize latency, which is done by Algorithm 8.

■

However, the problem is more complex when we consider different failure probabilities (*Failure Heterogeneous*). It is also more natural since we have different processors and there is no reason why they would have the same failure probability. Unfortunately for *Failure Heterogeneous*, we can exhibit for some problem instances an optimal solution in which the pipeline stages must be divided in several intervals. The complexity of the problem remains open, but we conjecture it is NP-hard.

Fully Heterogeneous Platforms

For *Fully Heterogeneous* platforms the bi-criteria problem with latency and failure probability constraints is NP-hard since the minimizing the latency on this kind of platform is already NP-hard (see Section 5.1.2).

5.3 Linear Program Formulation

We present here an integer linear program to compute the optimal interval-based bi-criteria mapping on *Fully Heterogeneous* platforms, respecting either a fixed latency or a fixed period. We consider a framework of n stages and p processors, plus two fictitious extra stages $\mathcal{S}_0$ and $\mathcal{S}_{n+1}$ respectively assigned to P_{in} and P_{out}. First we need to define a few variables.

- For $k \in [0..n+1]$ and $u \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}$, $x_{k,u}$ is a boolean variable equal to 1 if stage $\mathcal{S}_k$ is assigned to processor P_u; we let $x_{0,\mathsf{in}} = x_{n+1,\mathsf{out}} = 1$, and $x_{k,\mathsf{in}} = x_{k,\mathsf{out}} = 0$ for $1 \leq k \leq n$.
- For $k \in [0..n]$, $u, v \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}$ with $u \neq v$, $z_{k,u,v}$ is a boolean variable equal to 1 if stage $\mathcal{S}_k$ is assigned to P_u and stage S_{k+1} is assigned to P_v: hence $\mathsf{link}_{u,v} : P_u \to P_v$ is used for the communication between these two stages.
- If $k \neq 0$ then $z_{k,\mathsf{in},v} = 0$ for all $v \neq \mathsf{in}$ and if $k \neq n$ then $z_{k,u,\mathsf{out}} = 0$ for all $u \neq \mathsf{out}$.
- For $k \in [0..n]$ and $u \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}$, $y_{k,u}$ is a boolean variable equal to 1 if stages $\mathcal{S}_k$ and S_{k+1} are both assigned to P_u; we let $y_{k,\mathsf{in}} = y_{k,\mathsf{out}} = 0$ for all k, and $y_{0,u} = y_{n,u} = 0$ for all u.
- For $u \in [1..p]$, $\mathsf{first}(u)$ is an integer variable which denotes the first stage assigned to P_u; similarly, $\mathsf{last}(u)$ denotes the last stage assigned to P_u. Thus P_u is assigned the interval $[\mathsf{first}(u), \mathsf{last}(u)]$. Of course $1 \leq \mathsf{first}(u) \leq \mathsf{last}(u) \leq n$.
- T_{opt} is the variable to optimize, so depending on the objective function it corresponds either to the period or to the latency.

We list below the constraints that need to be enforced. For simplicity, we write $\sum_u$ instead of $\sum_{u \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}}$ when summing over all processors.

- First there are constraints for processor and link usage: every stage is assigned a processor, i.e., $\forall k \in [0..n+1]$, $\sum_u x_{k,u} = 1$.

- Every communication either is assigned a link or collapses because both stages are assigned to the same processor: $\forall k \in [0..n]$, $\sum_{u \neq v} z_{k,u,v} + \sum_u y_{k,u} = 1$.
- If stage $\mathcal{S}_k$ is assigned to P_u and stage $\mathcal{S}_{k+1}$ to P_v, then $\mathsf{link}_{u,v} : P_u \to P_v$ is used for this communication: $\forall k \in [0..n], \forall u, v \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}, u \neq v,\ x_{k,u} + x_{k+1,v} \leq 1 + z_{k,u,v}$.
- If both stages $\mathcal{S}_k$ and $\mathcal{S}_{k+1}$ are assigned to P_u, then $y_{k,u} = 1$: $\forall k \in [0..n], \forall u \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\},\ x_{k,u} + x_{k+1,u} \leq 1 + y_{k,u}$.
- If stage $\mathcal{S}_k$ is assigned to P_u, then necessarily $\mathsf{first}_u \leq k \leq \mathsf{last}_u$. We write this constraint as: $\forall k \in [1..n], \forall u \in [1..p],\ \mathsf{first}_u \leq k.x_{k,u} + n.(1 - x_{k,u})$ and $\forall k \in [1..n], \forall u \in [1..p],\ \mathsf{last}_u \geq k.x_{k,u}$.
- Furthermore, if stage $\mathcal{S}_k$ is assigned to P_u and stage $\mathcal{S}_{k+1}$ is assigned to $P_v \neq P_u$ (*i.e.*, $z_{k,u,v} = 1$) then necessarily $\mathsf{last}_u \leq k$ and $\mathsf{first}_v \geq k+1$ since we consider intervals. We write this constraint as: $\forall k \in [1..n-1], \forall u, v \in [1..p], u \neq v,\ \mathsf{last}_u \leq k.z_{k,u,v} + n.(1 - z_{k,u,v})$ and $\forall k \in [1..n-1], \forall u, v \in [1..p], u \neq v,\ \mathsf{first}_v \geq (k+1).z_{k,u,v}$.

The latency of the schedule is bounded by $\mathcal{L}$:

$$\sum_{u=1}^{p} \sum_{k=1}^{n} \left[\left(\sum_{t \neq u} \frac{\mathsf{comm}_{k-1}}{\mathsf{b}_{t,u}} z_{k-1,t,u} \right) + \frac{\mathsf{w}_k}{\mathsf{s}_u} x_{k,u} \right] + \left(\sum_{u \in [1..p] \cup \{\mathsf{in}\}} \frac{\mathsf{comm}_n}{\mathsf{b}_{u,\mathsf{out}}} z_{n,u,\mathsf{out}} \right) \leq \mathcal{L}$$

and $t \in [1..p] \cup \{\mathsf{in}, \mathsf{out}\}$.

There remains to express the period of each processor and to constrain it by $\mathcal{P}$: $\forall u \in [1..p]$,

$$\sum_{k=1}^{n} \left\{ \left(\sum_{t \neq u} \frac{\mathsf{comm}_{k-1}}{\mathsf{b}_{t,u}} z_{k-1,t,u} \right) + \frac{\mathsf{w}_k}{\mathsf{s}_u} x_{k,u} + \left(\sum_{v \neq u} \frac{\mathsf{comm}_k}{\mathsf{b}_{u,v}} z_{k,u,v} \right) \right\} \leq \mathcal{P}.$$

Finally, the objective function is either to minimize the period $\mathcal{P}$ respecting the fixed latency $\mathcal{L}$ or to minimize the latency $\mathcal{L}$ with a fixed period $\mathcal{P}$. So in the first case we fix $\mathcal{L}$ and set $T_{\mathsf{opt}} = \mathcal{P}$. In the second case $\mathcal{P}$ is fixed a priori and $T_{\mathsf{opt}} = \mathcal{L}$. With this mechanism the objective function reduces to minimizing T_{opt} in both cases.

Chapter 6

Case Study: Mapping the JPEG Encoder Pipeline onto a Cluster of Workstations

In this chapter we study the mapping of a particular pipeline application: we focus on the JPEG encoder (baseline process, basic mode). This image processing application transforms numerical pictures from any format into a standardized format called JPEG. This standard was developed almost 20 years ago to create a portable format for the compression of still images and new versions are created until now (see http://www.jpeg.org/). Meanwhile, several parallel algorithms have been proposed [50]. JPEG (and later JPEG 2000) is used for encoding still images in Motion-JPEG (later MJ2). These standards are commonly employed in IP-cams and are part of many video applications in the world of game consoles. Motion-JPEG (M-JPEG) has been adopted and further developed to several other formats, e.g., AMV (alternatively known as MTV) which is a proprietary video file format designed to be consumed on low-resource devices. The manner of encoding in M-JPEG and subsequent formats leads to a flow of still image coding, hence pipeline mapping is appropriate.

We consider the different steps of the encoder as a linear pipeline of stages, where each stage gets some input, has to perform several computations and transfers the output to the next stage. The corresponding mapping problem can be stated informally as follows: which stage to assign to which processor? We require the mapping to be interval-based, i.e., a processor is assigned an interval of consecutive stages. Two key optimization parameters emerge. On the one hand, we target a high throughput, or short period, in order to be able to handle as many images as possible per time unit. On the other hand, we aim at a short response time, or latency, for the processing of each image. These two criteria are antagonistic: intuitively, we obtain a high throughput with many processors to share the work, while we get a small latency by mapping many stages to the same processor in order to avoid the cost of inter-stage communications.

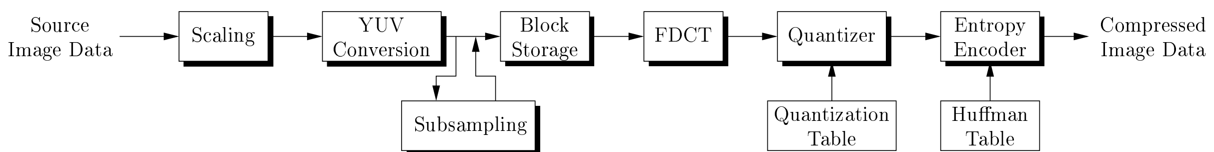


Figure 6.1: Steps of the JPEG encoding.

6.1 Principles of JPEG Encoding

Here we briefly present the mode of operation of a JPEG encoder (see [73] for further details). The encoder consists in seven pipeline stages, as shown in Fig. 6.1. In the first stage, the image is scaled to have a multiple of an 8x8 pixel matrix, and the standard even claims a multiple of 16x16. In the next stage a color space conversion is performed from the RGB to the YUV-color model. The sub-sampling stage is an optional stage, which, depending on the sampling rate, reduces the data volume: as the human eye can distinguish luminosity more easily than color, the chrominance components are sampled more rarely than the luminance components. Admittedly, this leads to a loss of data. The last preparation step consists in the creation and storage of so-called MCUs (Minimum Coded Units), which correspond to 8x8 pixel blocks in the picture. The next stage is the core of the encoder. It performs a Fast Discrete Cosine Transformation (FDCT) (eg. [75]) on the 8x8 pixel blocks which are interpreted as a discrete signal of 64 values. After the transformation, every point in the matrix is represented as a linear combination of the 64 points. The quantizer reduces the image information to the important parts. Depending on the quantization factor and quantization matrix, irrelevant frequencies are reduced. Thereby quantization errors can occur, that are remarkable as quantization noise or block generation in the encoded image. The last stage is the entropy encoder, which performs a modified Huffman coding.

6.2 Heuristics

In this section several polynomial heuristics for *Communication Homogeneous* platforms are presented. We restrict to such platforms because clusters made of different-speed processors interconnected by either plain Ethernet or a high-speed switch constitute the typical experimental platforms in most academic or industry research departments. Moreover, this bi-criteria problem is already NP-hard because of the period minimization problem. Note that it would be much more difficult to design efficient heuristics for *Fully Heterogeneous* platforms since the latency minimization problem also becomes NP-hard. In fact, it would become hard to predict the latency of a mapping before the mapping is entirely known.

In the following, we denote by n the number of stages, and by p the number of processors. Note that the code for all these heuristics can be found on the Web[1].

6.2.1 Minimizing Latency for a Fixed Period

In the first set of heuristics, the period is fixed a priori, and we aim at minimizing the latency while respecting the prescribed period. All the following heuristics sort processors by non-increasing speed, and start by assigning all the stages to the first (fastest) processor in the list. This processor becomes *used.*

- **P1 2-Split mono: 2-Splitting mono-criterion** – At each step, we select the used processor j with the largest period and we try to split its stage interval, giving some stages to the next fastest processor j' in the list (not yet used). This can be done by splitting the interval at any place, and either placing the first part of the interval on j and the remainder on j', or the other way round. The solution which minimizes $\max(period(j), period(j'))$ is

[1] http://graal.ens-lyon.fr/~vsonigo/code/multicriteria/

chosen if it is better than the original solution. Splitting is performed as long as we have not reached the fixed period or until we cannot improve the period anymore.

- **P2 3-Split mono: 3-Splitting mono-criterion** – At each step we select the used processor j with the largest period and we split its interval into three parts. For this purpose we try to map two parts of the interval on the next pair of fastest processors in the list, j' and j'', and to keep the third part on processor j. Testing all possible permutations and all possible positions where to cut, we choose the solution that minimizes $\max(period(j), period(j'), period(j''))$.

- **P3 3-Split bi: 3-Splitting bi-criteria** – In this heuristic the choice of where to split is more elaborated: it depends not only of the period improvement, but also of the latency increase. Using the same splitting mechanism as in **P2 3-Split mono**, we select the solution that minimizes $\max_{i\in\{j,j',j''\}}(\frac{\Delta latency}{\Delta period(i)})$. Here $\Delta latency$ denotes the difference between the global latency of the solution before the split and after the split. In the same manner $\Delta period(i)$ defines the difference between the period before the split (achieved by processor j) and the new period of processor i.

6.2.2 Minimizing Period for a Fixed Latency

In this second set of heuristics, latency is fixed, and we try to achieve a minimum period while respecting the latency constraint. As in the heuristics described above, first of all we sort processors according to their speed and map all stages on the fastest processor. The approach used here is the converse of the heuristics where we fix the period, as we start with an optimal solution concerning latency. Indeed, at each step we downgrade the solution with respect to its latency but improve it regarding its period.

- **L1 2-Split mono: 2-Splitting mono-criterion** – This heuristic uses the same method as **P1 2-Split mono**, with a different break condition. Here splitting is performed as long as we do not exceed the fixed latency, still choosing the solution that minimizes $\max(period(j), period(j'))$.

- **L2 2-Split bi: 2-Splitting bi-criteria** – This variant of the splitting heuristic works similarly to **L1 2-Split mono**, but at each step it chooses the solution which minimizes $\max_{i\in\{j,j'\}}(\frac{\Delta latency}{\Delta period(i)})$ while the fixed latency is not exceeded.

6.3 Experimental Results

In this section we present a performance evaluation of our polynomial heuristics. Section 6.3.1 evaluates the heuristics in a general context, i.e., using randomly generated workflow applications. Then Section 6.3.2 is dedicated to the evaluation of our heuristics for a real world application, namely the JPEG encoder application pipeline.

6.3.1 General Experiments

Several general experiments have been conducted in order to assess the performance of the heuristics described in Section 6.2. First we describe the experimental setting, then we report the results, and finally we provide a summary.

Experimental Setting

We have generated a set of random applications with $n \in \{5, 10, 20, 40\}$ stages and a set of random *Communication Homogeneous* platforms with $p = 10$ or $p = 100$ processors. In all the experiments, we fix $\mathsf{b} = 10$ for the link bandwidths. Moreover, the speed of each processor is randomly chosen as an integer between 1 and 20. We keep the latter range of variation throughout the experiments, while we vary the range of the application parameters from one set of simulations to the other. Indeed, although there are four categories of parameters to play with, i.e., the values of comm, w, s and b, we can see from equations (4.1) and (4.3) that only the relative ratios $\frac{\mathsf{comm}}{\mathsf{b}}$ and $\frac{\mathsf{w}}{\mathsf{s}}$ have an impact on the performance. Each experimental value reported in the following has been calculated as an average over 50 randomly chosen application/platform pairs. For each of these pairs, we report the performance of the six heuristics described in Section 6.2.

We report four sets of experiments conducted both for $p = 10$ and $p = 100$ processors. For each experiment, we vary some key application/platform parameter to assess the impact of this parameter on the performance of the heuristics. The first two experiments deal with applications where communications and computations have the same order of magnitude, and we study the impact of the degree of heterogeneity of the communications, i.e., of the variation range of the comm parameter:

- **(E1): balanced communication/computation, and homogeneous communications.** In the first set of experiments, the application communications are homogeneous, we fix $\mathsf{comm}_i = 10$ for $i = 0..n$. The computation time required by each stage is randomly chosen between 1 and 20. Thus, communications and computations are balanced within the application.

- **(E2): balanced communications/computations, and heterogeneous communications.** In the second set of experiments, the application communications are heterogeneous, chosen randomly between 1 and 100. Similarly to Experiment 1, the computation time required by each stage is randomly chosen between 1 and 20. Thus, communications and computations are still relatively balanced within the application.

The last two experiments deal with imbalanced applications: the third experiment assumes large computations (large value of the w to comm ratio), and the fourth one reports results for small ones (small value of the w to comm ratio):

- **(E3): large computations.** In this experiment, the applications are much more demanding on computations than on communications, making communications negligible with respect to computation requirements. We choose the communication time between 1 and 20, while the computation time of each application is chosen between 10 and 1000.

- **(E4): small computations.** The last experiment is the opposite to Experiment 3 since computations are now negligible compared to communications. The communication time is still chosen between 1 and 20, but the computation time is now chosen between 0.01 and 10.

Results

Results for the entire set of experiments can be found on the Web at
`http://graal.ens-lyon.fr/~vsonigo/code/multicriteria/`.
In the following we only present the most significant plots.

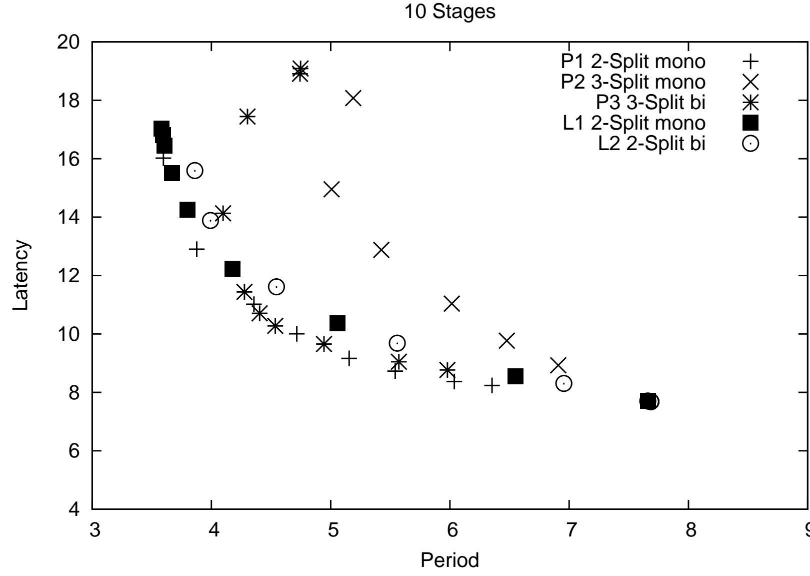


Figure 6.2: $p = 10$ - (E1) Balanced communications/computations, and homogeneous communications. 10 stages.

For each heuristic with a fixed period, we compute for a set of periods the resulting latency, and we plot the latency as a function of the period. If a small period is chosen, the heuristic may fail to find a feasible solution. In this case, there is no corresponding point in the plot and the largest value of such period is reported in the failure thresholds table (see Table 6.1).

Similarly, the values of resulting period obtained for the heuristics with a fixed latency are plotted, reporting the period as a function of the latency. In this case, the heuristic may fail to find a solution if the fixed latency is too small, and latency thresholds are detailed in Table 6.1.

This way, both categories of heuristics can be plotted into the same figure for a given experimental setup, and all heuristics can be compared one to each other.

With $p = 10$ processors

For (E1) we see that all heuristics follow the same curve shape. they achieve small period times at the price of long latencies and then seem to converge to a somewhat shorter latency. We observe that the simplest splitting heuristics perform very well: **P1 2-Split mono** and **L1 2-Split mono** achieve the best period, and **P1 2-Split mono** has the lower latency. **L2 2-Split bi** performs poorly in comparison. **P2 3-Split mono** and **L2 2-Split bi** cannot keep up with the other heuristics (but the latter achieves better results than the former). In the middle range of period values, **P3 3-Split bi** achieves comparable latency values with those of **P1 2-Split mono**.

For (E2), we see that **P1 2-Split mono** outperforms the other heuristics almost everywhere with the following exception: with 40 stages and a large fixed period, **P3 3-Split bi** obtains the better results. The period times of **P1 2-Split mono** and **P3 3-Split bi** are the best. We observe that the competitiveness of **P3 3-Split bi** increases with the increase of the number of stages. **L1 2-Split mono** achieves period values just as small as **P1 2-Split mono** but the corresponding latency is higher and once again it performs better than its bi-criteria counterpart **L2 2-Split bi**. The poorest results are obtained by **P2 3-Split mono**.

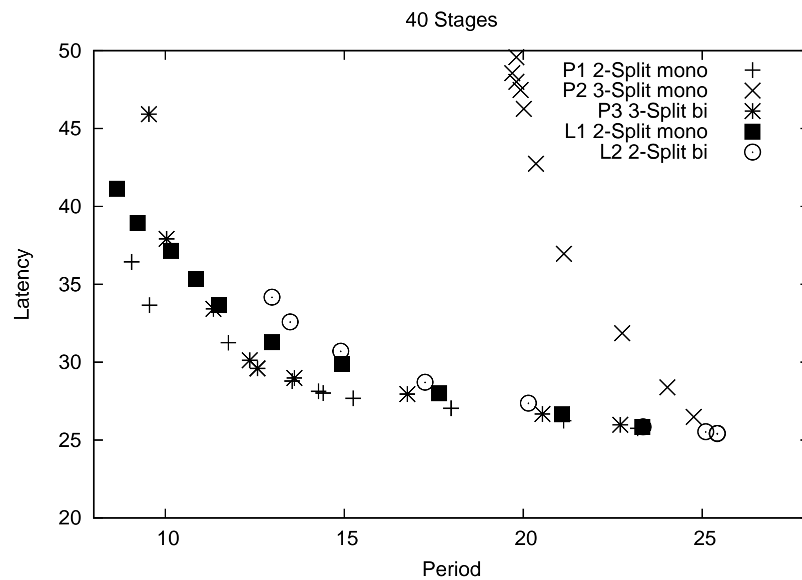


Figure 6.3: $p = 10$ - (E1) Balanced communications/computations, and homogeneous communications. 40 stages.

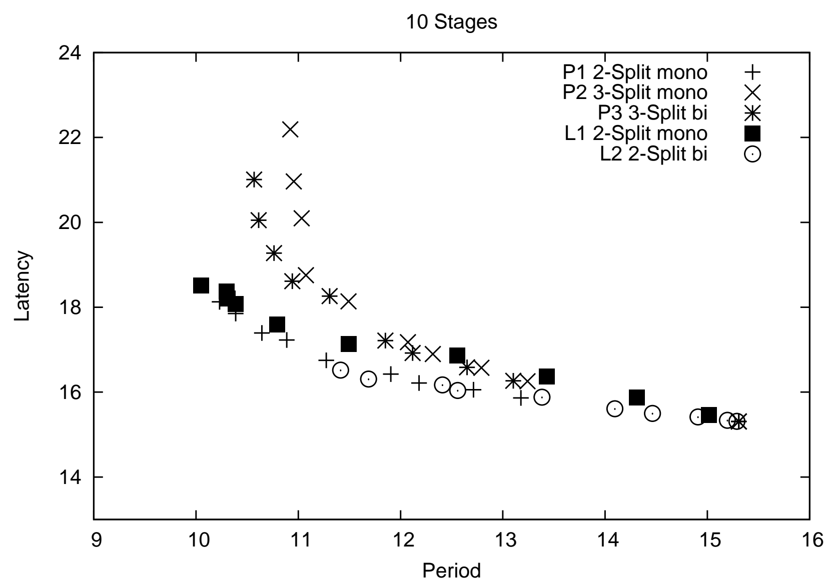


Figure 6.4: $p = 10$ - (E2) Balanced communications/computations, and heterogeneous communications. 10 stages.

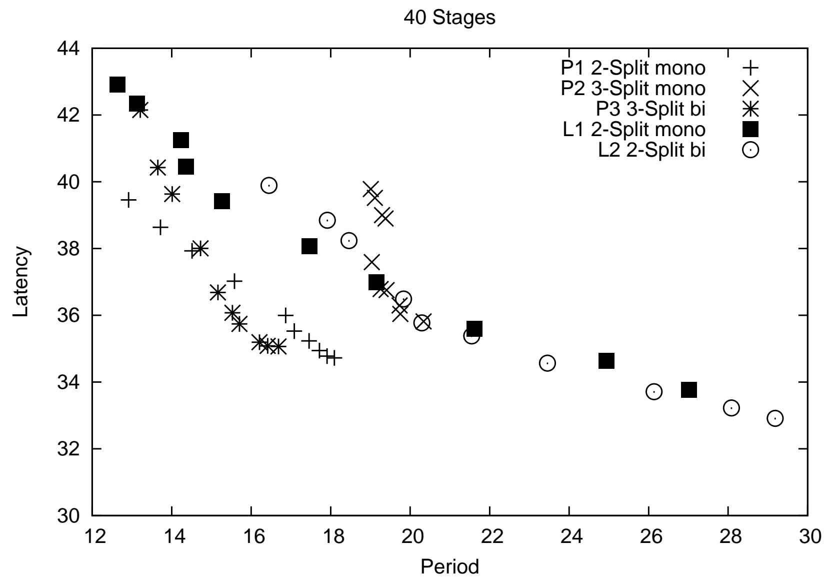


Figure 6.5: $p = 10$ - (E2) Balanced communications/computations, and heterogeneous communications. 40 stages.

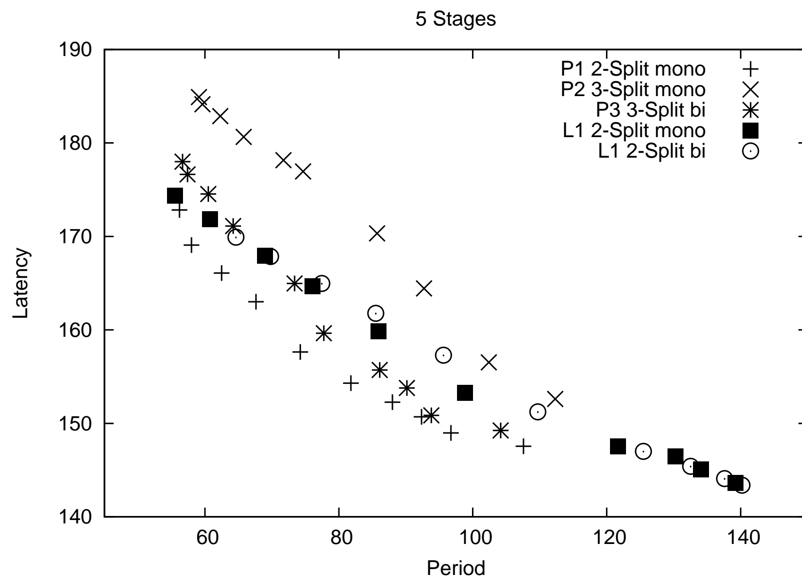


Figure 6.6: $p = 10$ - (E3) Large computations. 5 stages.

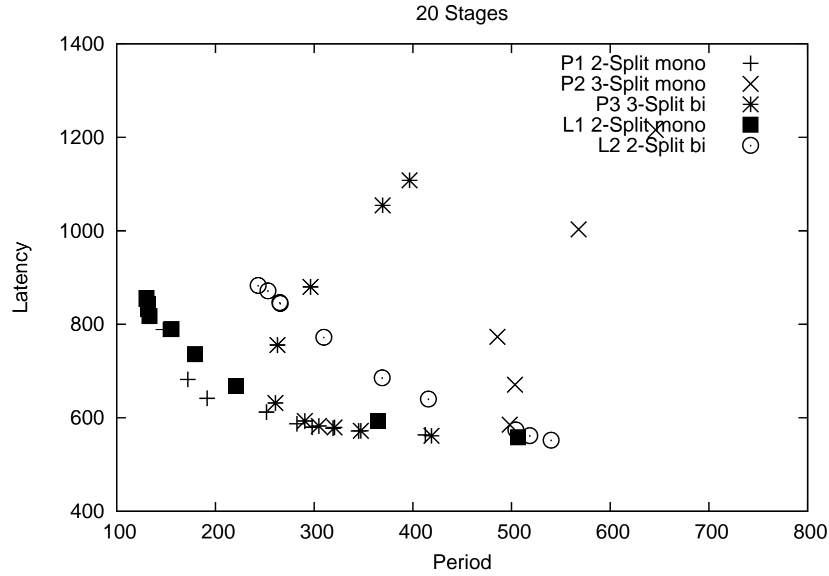


Figure 6.7: $p = 10$ - (E3) Large computations. 20 stages.

The results of (E3) are much more scattered than in the other experiments (E1, E2 and E4) and this difference even increases with rising n. When $n = 5$, the results of the different heuristics are almost parallel so that we can state the following hierarchy: **P1 2-Split mono**, **P3 3-Split bi**, **L1 2-Split mono**, **L2 2-Split bi** and finally **P2 3-Split mono**. **P1 2-Split mono** and **P3 3-Split bi** achieve very good results concerning period durations. On the contrary, **P2 3-Split mono** exploses its period and latency times. **P3 3-Split bi** loses its second position for small period times compared to **L1 2-Split mono**, but when period times are higher it recovers its position in the hierarchy.

In (E4), **P2 3-Split mono** performs the poorest. Nevertheless the gap is smaller than in (E3) and for high period times and $n \geq 20$, its latency is comparable to those of the other heuristics. For $n \geq 20$, **P3 3-Split bi** achieves for the first time the best results. When $n = 5$, **L2 2-Split bi** achieves the best latency, but the period values are not competitive with **P1 2-Split mono** and **L1 2-Split mono**, which obtain the smallest periods (for slightly higher latency times).

In Table 6.1 the **failure thresholds** of the different heuristics are shown. We denote by failure threshold the largest value of the fixed period or latency for which the heuristic was not able to find a solution. We state that **P1 2-Split mono** has the smallest failure thresholds whereas **P2 3-Split mono** has the highest values. Surprisingly the failure thresholds (for fixed latencies) of the heuristics **L1 2-Split mono** and **L2 2-Split bi** are the same, but their performance differs enormously as stated in the different experiments.

With $p = 100$ processors

Many results are similar with $p = 10$ and $p = 100$ processors, thus we only report the main differences. In particular, the failure thresholds table is not included since it displays similar results.

First we observe that both periods and latencies are lower with the increasing number of processors. This is easy to explain, as all heuristics always choose fastest processors first, and

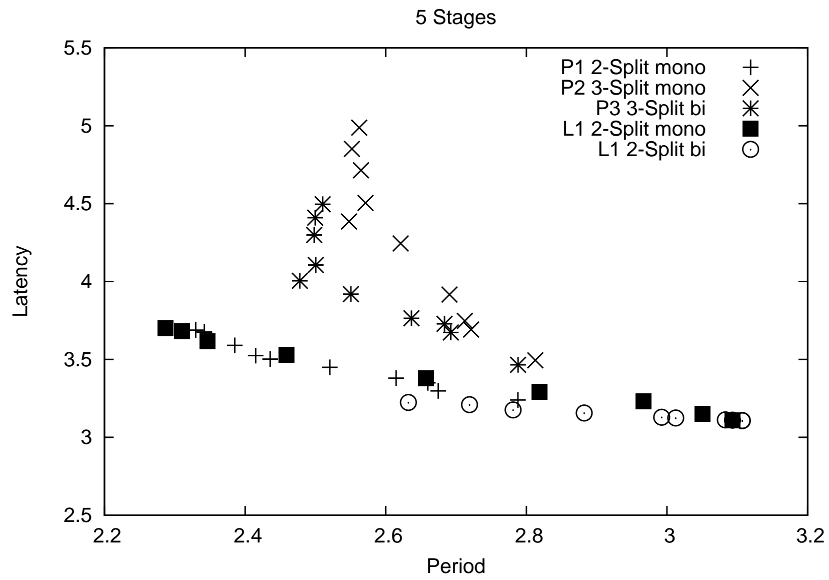


Figure 6.8: $p = 10$ - (E4) Small computations. 5 stages.

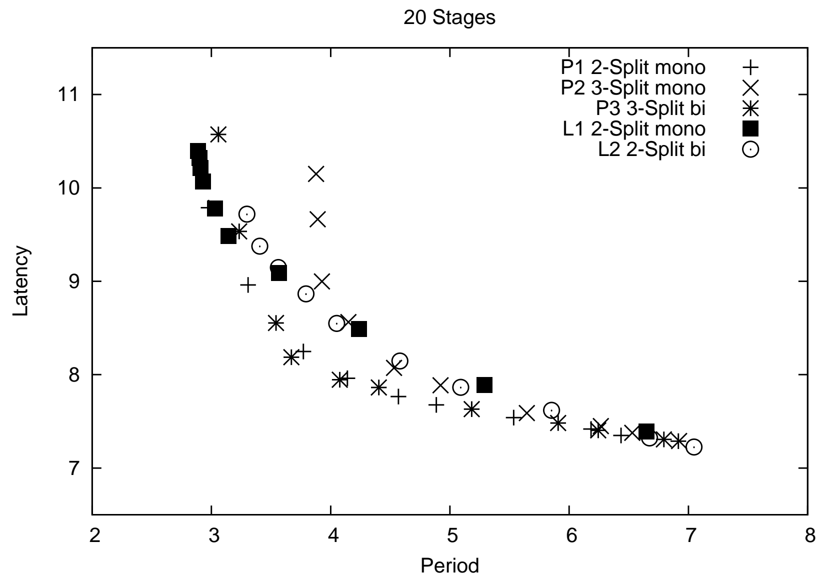


Figure 6.9: $p = 10$ - (E4) Small computations. 20 stages.

Table 6.1: Failure thresholds with $p = 10$.

Exp.	Heur.	Number of stages				Exp.	Heur.	Number of stages			
		5	10	20	40			5	10	20	40
E1	P1	3.0	3.3	5.0	5.0	E3	P1	50.0	70.0	100.0	250.0
	P2	3.0	4.0	5.0	5.0		P2	50.0	90.0	250.0	400.0
	P3	3.3	3.3	6.0	10.0		P3	100.0	140.0	300.0	650.0
	L1	4.5	6.0	13.0	25.0		L1	140.0	270.0	500.0	1000.0
	L2	4.5	6.0	13.0	25.0		L2	140.0	270.0	500.0	1000.0
E2	P1	9.7	10.0	11.0	11.0	E4	P1	2.2	2.3	2.3	2.3
	P2	10.0	10.0	11.0	11.0		P2	2.4	2.7	3.0	4.0
	P3	11.3	11.0	13.0	15.0		P3	2.8	2.7	3.0	4.0
	L1	11.7	15.0	22.0	32.0		L1	3.0	4.0	7.0	11.0
	L2	11.7	15.0	22.0	32.0		L2	3.0	4.0	7.0	11.0

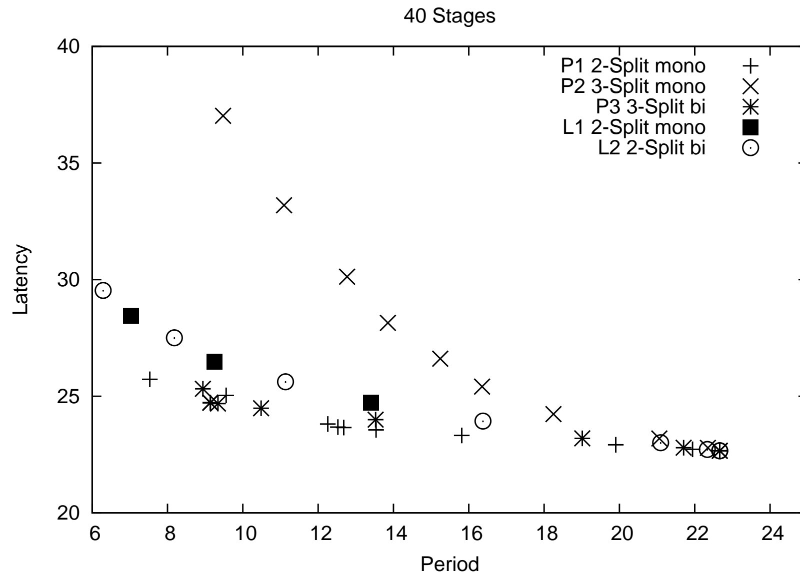


Figure 6.10: (E1) Extension to 100 processors, balanced communications/computations. 40 stages, hom. comms.

there is much more choice with $p = 100$. All heuristics keep their general behavior, i.e., their curve characteristics. But the relative performance of some heuristics changes dramatically. The results of **P2 3-Split mono** are much better, and we do get adequate latency times (compare Figures 6.3 and 6.10). Furthermore the multi-criteria heuristics turn out to be much more performing. An interesting example can be seen in Figure 6.11: all multi-criteria heuristics outperform their mono-criterion counterparts, even **L2 2-Split bi**, which never had a better performance than **L1 2-Split mono** when $p = 10$.

In the case of imbalanced communications/computations, we observe that all heuristics achieve almost the same results (see Figure 6.12.) The performance of **P3 3-Split bi** depends on the number of stages. In general, **P1 2-Split mono** is better than **P3** when $n \leq 10$, but for $n \geq 20$, **P3** owns the second position after **L2 2-Split bi** and even performs best in the configuration small computations, $n = 40$ (see Figure 6.13).

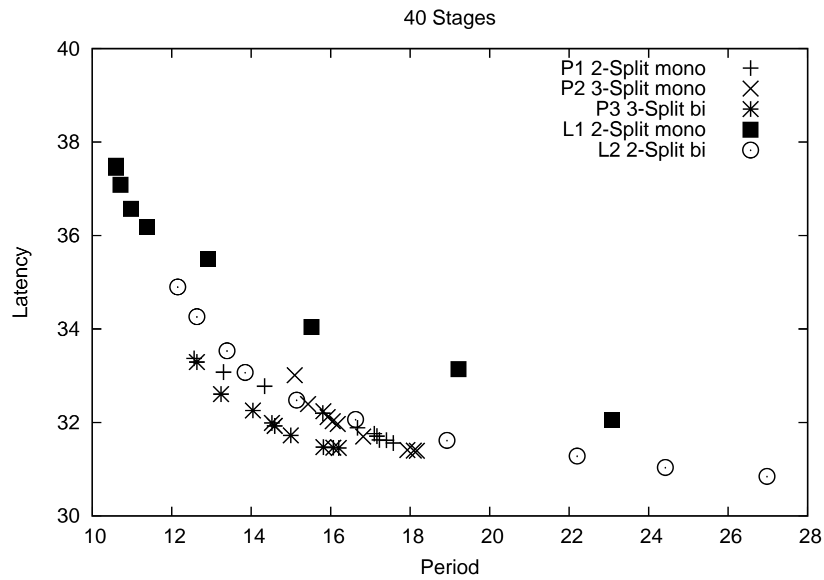


Figure 6.11: (E2) Extension to 100 processors, balanced communications/computations. 40 stages, het. comms.

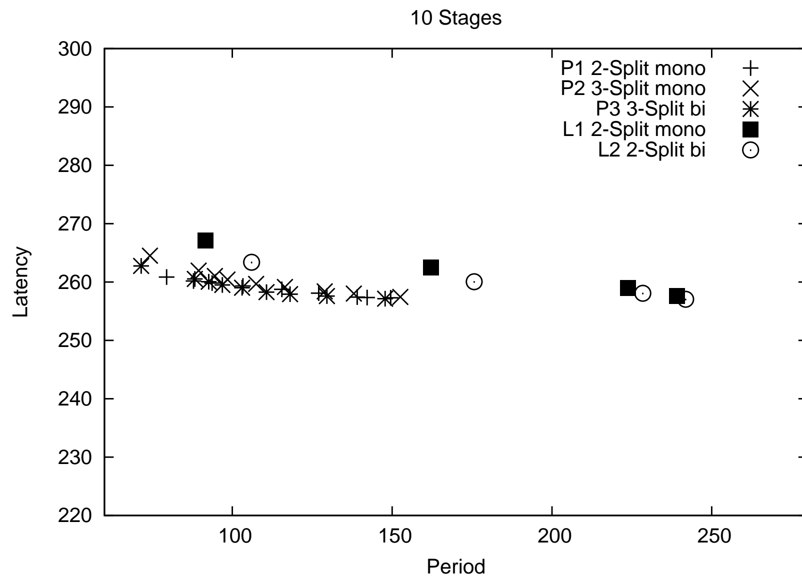


Figure 6.12: (E3) Extension to 100 processors, imbalanced communications/computations. 10 stages, large computations.

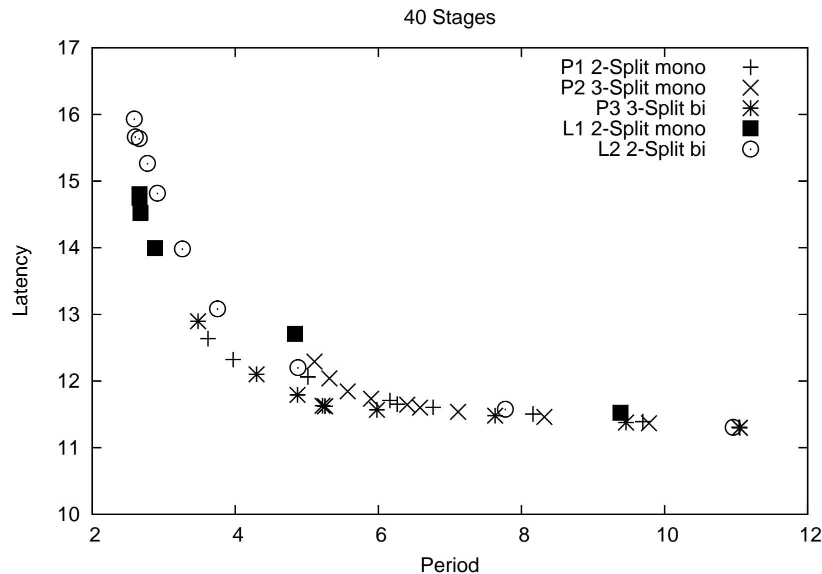


Figure 6.13: (E4) Extension to 100 processors, imbalanced communications/computations. 40 stages, small computations.

Comparison to the LP solution

In order to assess the absolute performance of our heuristics, we compared their solution to the optimal solution returned by the linear program of Section 5.3. We took the 500 set of parameters considered in (E2) with 40 stages, and set up a time limit to one hour to solve each problem instance with the LP solver. Within this time limit, 94 instances (out of 500) were solved for a fixed period, and only 8 for a fixed latency.

Comparing the solution returned by the heuristics to the optimal solution for those cases for which we obtained results, the conclusion is that the **2-Split** heuristic with fixed period (**P1** behaves best, being on average at less than 5% of the optimal. In many cases, these heuristics return the optimal solution, and otherwise their solution is close to the optimal. The **3-Split** heuristics (**P2** and **P3**) are slightly behind, with 6%, thus still very competitive.

For **L1 2-Split mono** and **L2 2-Split bi**, we could not produce statistics because of the low number of solved instances, but we found the optimal solution in 5 of the 8 cases. In the three remaining cases, however, heuristic results are further from the optimal, leading to periods up to 60% away from the optimal solution. The detailed values of the LP and the heuristics can be found on the Web[2].

Summary

Overall we conclude that the performance of bi-criteria heuristics versus mono-criterion heuristics highly depends on the number of available processors. For a small number of processors, the simple splitting technique which is used in **P1 2-Split mono** and **L1 2-Split mono** is very competitive as it almost always minimizes the period with acceptable latency values. The bi-criteria splitting **L2 2-Split bi** does not provide convincing results, and both **3-Split** heuristics do not achieve the expected performance. However, when increasing the number of available

[2] http://graal.ens-lyon.fr/~vsonigo/code/multicriteria/

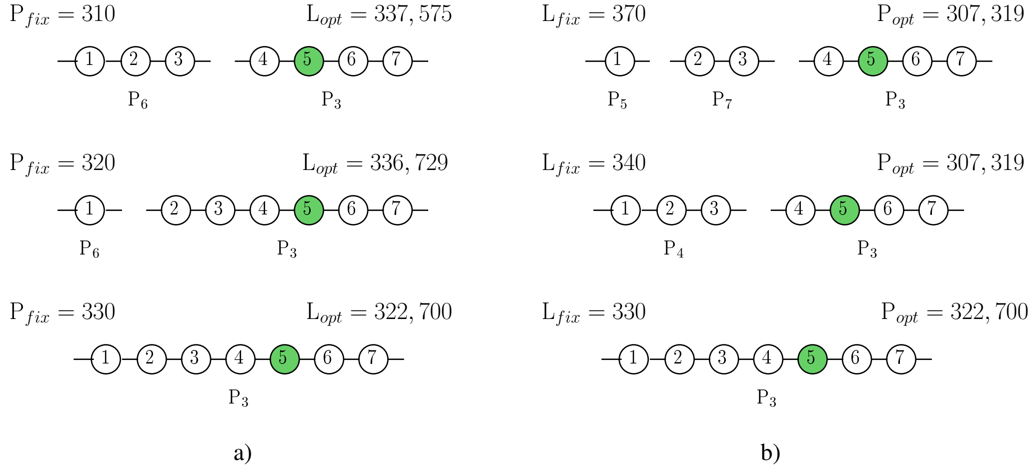


Figure 6.14: LP solutions strongly depend on fixed initial parameters.

processors, we observe a significant improvement of the behavior of bi-criteria heuristics. **L2 2-Split bi** turns out to outperform the mono-criterion version. Finally, both 3-Splitting heuristics perform much better and **P3 3-Split bi** finds its slot.

To conclude, we point out that in most of the cases for which we are able to compute the optimal solution thanks to the linear program, our heuristics are very close to this optimal solution (or even returning the optimal), in particular for the heuristics with a fixed period. Heuristics with a fixed latency are slightly further from the optimal. Note that this is a very promising result, given that running all heuristics for all parameters takes less than one minute, while the LP solver does not succeed to find the solution within one hour for most problem instances.

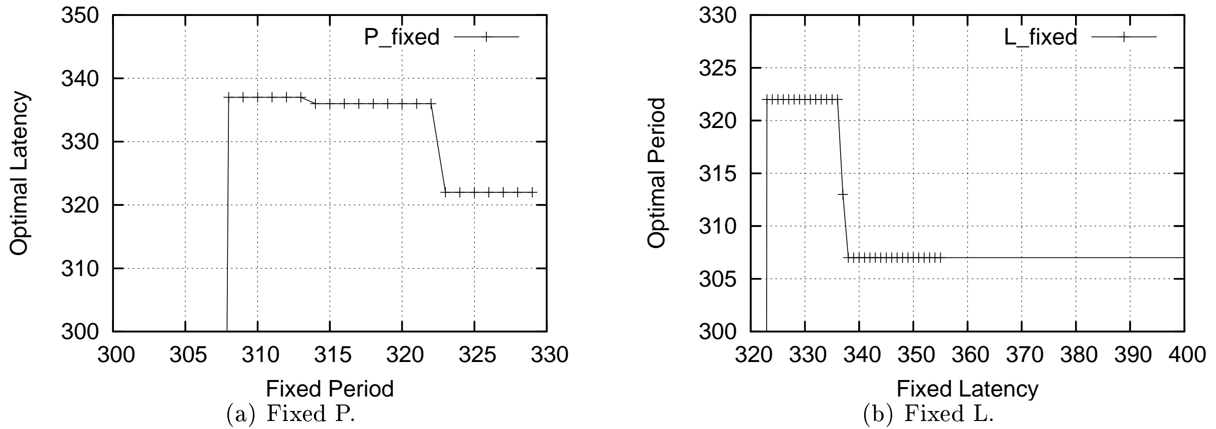
6.3.2 Experiments and Simulations for the JPEG encoder

In the following experiments, we study the mapping of the JPEG application onto clusters of workstations.

Influence of fixed parameters

In the first test series, we examine the influence of fixed parameters on the solution of the linear program. As shown in Figure 6.14, the division into intervals is highly dependant of the chosen fixed value. The optimal solution to minimize the latency (without any additional constraints) obviously consists in mapping the whole application pipeline onto the fastest processor. As expected, if the period fixed in the linear program is not smaller than the latter optimal mono-criterion latency, this solution is chosen. Decreasing the value for the fixed period imposes to split the stages among several processors, until no more solution can be found.

Figure 6.14(a) shows the division into intervals for a fixed period. A fixed period of $\mathcal{P} = 330$ is sufficiently high for the whole pipeline to be mapped onto the fastest processor, whereas smaller periods lead to splitting into intervals. We would like to mention that for a period fixed to 300, there exists no solution anymore. The counterpart (with fixed latency) can be found in Figure 6.14(b). Note that the first two solutions find the same period, but for a different latency. The first solution has a high value for latency, which allows more splits, hence larger



(a) Fixed P. (b) Fixed L.

Figure 6.15: Bucket behavior of LP solutions.

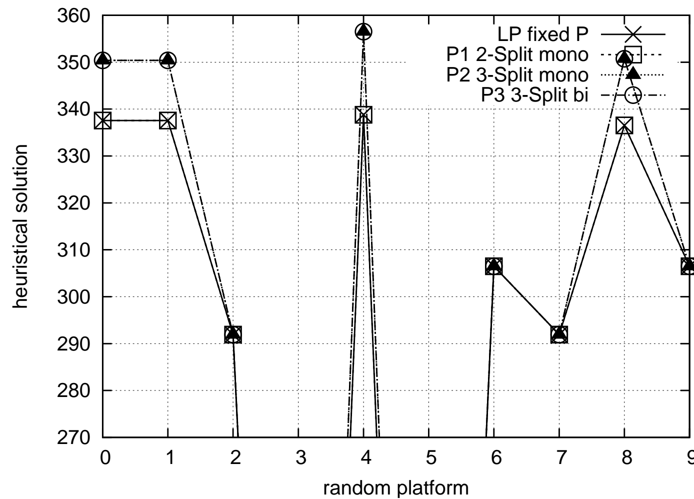


Figure 6.16: Behavior of the heuristics (comparing to LP solution). Fixed P = 310.

communication costs. Comparing the last lines of Figure 6.14(a) and (b), we state that both solutions are the same, and we have $\mathcal{P} = \mathcal{L}$. Finally, expanding the range of the fixed values, a sort of bucket behavior becomes apparent: Increasing the fixed parameter has in a first time no influence, the LP still finds the same solution until the increase crosses an unknown bound and the LP can find a better solution. This phenomenon is shown in Figure 6.15.

Assessing heuristic performance

The comparison of the solution returned by the LP program, in terms of optimal latency respecting a fixed period (or the converse) with the heuristics is shown in Figures 6.16 and 6.17. The implementation is fed with the parameters of the JPEG encoding pipeline and computes the mapping on 10 randomly created platforms with 10 processors. On platforms 3 and 5, no valid solution can be found for the fixed period.

We point out that all solutions, LP and heuristics, always keep the stages 4 to 7 together (see Figure 6.14 for an example). As stage 5 (DCT) is the most costly in terms of computation,

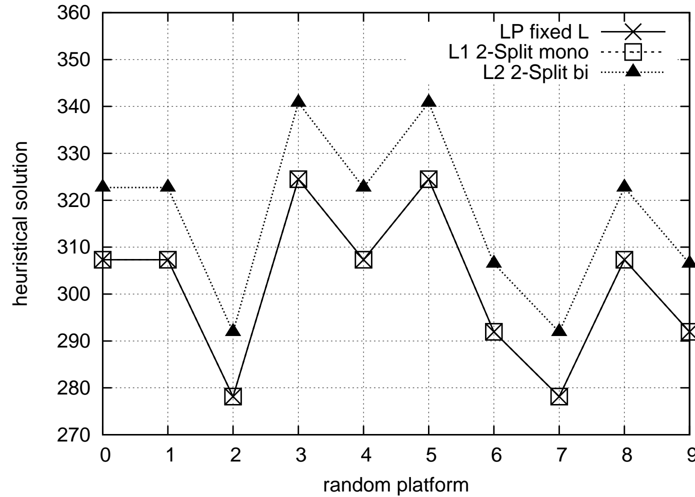


Figure 6.17: Behavior of the heuristics (comparing to LP solution). Fixed L = 370.

the interval containing these stages is responsible for the period of the whole application. Finally, in the comparative study **P1 2-Split mono** always finds the optimal latency for a fixed period and we therefore recommend this heuristic for latency optimization. In the case of period minimization for a fixed latency, then **L1 2-Split mono** is to use, as it always finds the LP solution in the experiments. This is a striking result, especially given the fact that the LP integer program may require a long time to compute the solution (up to 11389 seconds in our experiments), while the heuristics always complete in less than a second, and find the corresponding optimal solution.

MPI simulations on a cluster

This last experiment performs a JPEG encoding simulation. All simulations are made on a cluster of homogeneous Optiplex GX 745 machines with an Intel Core 2 Duo 6300 of 1,83Ghz. Heterogeneity is enforced by increasing and decreasing the number of operations a processor has to execute. The same holds for bandwidth capacities. For simplicity we use a MPI program whose stages have the same communication and computation parameters as the JPEG encoder, but we do not encode real images (hence the name simulation, although we use an actual implementation with MPICH).

In this experiment the same random platforms with 10 processors and fixed parameters as in the theoretical experiments are used. We measured the latency of the simulation, even for the heuristics of fixed latency, and computed the average over all random platforms. Figure 6.18 compares the average of the theoretical results of the heuristics to the average simulative performance, and the simulative behavior nicely mirrors the theoretical behavior.

Summary

We conclude that the general behavior of the heuristics in the special case of the JPEG encoder pipeline mainly reflects the observations made in the general case. The 2-Split technique seems to be a very powerful mechanism for bi-criteria optimization with a small number of processors, and always returns the optimal solution for the JPEG encoder.

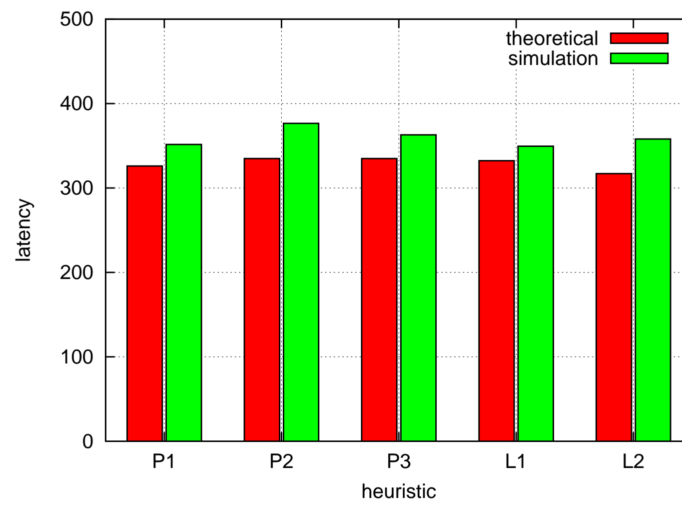


Figure 6.18: MPI simulation results - Simulative latency.

Part III

Complex Streaming Applications

Chapter 7

Introduction

This part deals with a third mapping problem, which can be viewed as a combination of the previous two problems. We consider the execution of applications structured as trees of operators, where the leaves of the tree correspond to basic data objects that are distributed over servers in a distributed network. Each internal node in the tree denotes the aggregation and combination of the data from its children, which in turn generates new data that is used by the node's parent. The computation is complete when all operators have been applied up to the root node, thereby producing a final result. We consider the scenario in which the basic data objects are constantly being updated, meaning that the tree of operators must be applied continuously. The goal is to produce final results at some desired rate. This problem is called *stream processing* [7] and arises in several domains.

An important domain of application is the acquisition and refinement of data from a set of sensors [65, 48, 15]. For instance, [65] outlines a video surveillance application in which the sensors are cameras located at different locations over a geographical area. The goal of the application could be to identify monitored areas in which there is significant motion between frames, particular lighting conditions, and correlations between the monitored areas. This can be achieved by applying several operators (e.g., filters, pattern recognition) to the raw images, which are produced/updated periodically. Another example arises in the area of network monitoring [26, 71, 24]. In this case routers produce streams of data pertaining to forwarded packets. More generally, stream processing can be seen as the execution of one of more "continuous queries" in the relational database sense of the term (e.g., a tree of join and select operators). A continuous query is applied continuously, i.e., at a reasonably fast rate, and returns results based on recent data generated by the data streams. Many authors have studied the execution of continuous queries on data streams [8, 45, 18, 56, 43].

In practice, the execution of the operators must be distributed over the network. In some cases the servers that produce the basic objects may not have the computational capability to apply all operators. Besides, objects must be combined across devices, thus requiring network communication. Although a simple solution is to send all basic objects to a central compute server, it often proves unscalable due to network bottlenecks. Also, this central server may not be able to meet the desired target rate for producing results due to the sheer amount of computation involved. The alternative is then to distribute the execution by mapping each node in the operator tree to one or more servers in the network, including servers that produce and update basic objects and/or servers that are only used for applying operators. One then talks of *in-network stream-processing*. Several in-network stream-processing systems have been

developed [2, 21, 37, 20, 51, 71, 19, 47]. These systems all face the same question: where should operators be mapped in the network?

The operator-mapping problem for in-network stream processing was studied in [65, 54, 5]. Most relevant to our work is the recent work of Pietzuch et al. [54], in which the problem is studied for an ad-hoc objective function that trades off application delay and network bandwidth consumption.

In Chapter 8 we study a more general objective function for a single application. We enforce the constraint that the rate at which final results are produced, or *throughput*, is above a given threshold. This corresponds to a Quality of Service (QoS) requirement, which is almost always desirable in practice (e.g., up-to-date results of continuous queries must be available at a given frequency). Basic objects may be replicated at multiple locations, i.e., available and updated at these locations. In terms of the computing platform we consider a "constructive" scenario: either the user can build the platform from scratch using off-the-shelf components, or computing and network units are rented by a cloud provider (e.g. [1]). Our goal is to construct a distributed network dedicated to the given application, which minimizes the monetary cost while ensuring that the desired throughput is achieved.

In Chapter 9 we address the operator-mapping problem for *multiple concurrent in-network stream-processing applications*. Instead of one single application, as in Chapter 8, we focus on multiple concurrent applications that contend for the servers. In this case each application has its own QoS requirement and the goal is to meet them all. A clear opportunity for higher performance with a reduced resource consumption is to reuse common sub-expression between operator trees when applications share basic objects [52]. Furthermore we study the problem in a "non-constructive" scenario, i.e., we are given a set of compute and network elements, and we attempt to use as few resources as possible while meeting QoS requirements. We restrict our study to trees of operators that are general binary trees and discuss relevant special cases (e.g., left-deep trees [38]). We consider target platforms that are either fully homogeneous, or with a homogeneous network but heterogeneous servers, or fully heterogeneous.

Chapter 8

In-Network Stream Processing

In this chapter we consider the operator mapping problem for in-network stream processing of single applications. Our aim is to provide the user a set of processors that should be bought or rented in order to ensure that the application achieves a minimum steady-state throughput, and with the objective of minimizing platform cost.

Our contributions are as follows: (i) we formalize the operator-placement problem; (ii) we establish complexity results (all problems turn out to be NP-complete); (iii) we propose several polynomial heuristics; (iv) we compare heuristics through extended simulations, and assess their absolute performance.

8.1 Models

8.1.1 Application Model

We consider an application that can be represented as a set of operators $\mathcal{N} = \{n_1, n_2, \ldots\}$ arranged as a binary tree, as shown in Figure 9.1. Operations are initially performed on basic objects, which are made available and continuously updated at given locations in a distributed network. We denote the set of basic objects, which are leaves of the tree, by $\mathcal{OB} = \{ob_1, ob_2, \ldots\}$. Several leaves may correspond to the same object, as illustrated in the figure. Internal nodes represent operator computations. For an operator n_i we define $Leaf(i)$ as the index set of the basic objects needed for the computation of n_i, if any, $Child(i)$ as the index set of the node's children in $\mathcal{N}$, if any, and $Parent(i)$ as the index of the node's parent in $\mathcal{N}$, if it exists. We have the constraint that $|Leaf(i)| + |Child(i)| \leq 2$ because the tree is binary. All functions above are extended to sets of nodes: $f(I) = \cup_{i \in I} f(i)$, where I is an index set and f is $Leaf$, $Child$ or $Parent$. If $|Leaf(i)| \geq 1$, then operator n_i needs at least one basic object for its computation. We call such an operator an *al-operator* (for "almost leaf").

The application must be executed so that it produces final results, where each result is generated by executing the whole operator tree once, at a target rate. We call this rate the application *throughput*. Each operator $n_i \in \mathcal{N}$ must compute (intermediate) results at a rate at least as high as the target application throughput. Conceptually, a server executing an operator consists of two concurrent threads that run in steady-state. One thread periodically downloads the most recent copies of the basic objects corresponding to the operator's leaf children, if any. For our example tree in Figure 8.1(a), n_1 needs to download ob_1 and ob_2 while n_2 downloads only ob_1 and n_5 does not download any basic object. Note that these downloads may simply amount to constant streaming of data from sources that generate data streams. Each download

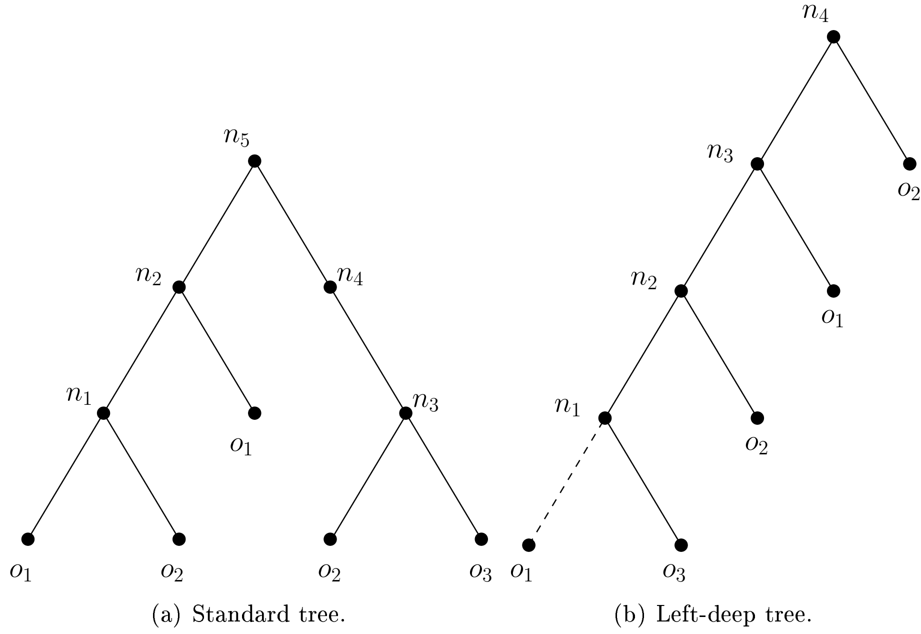


Figure 8.1: Examples of applications structured as a binary tree of operators.

has a prescribed cost in terms of bandwidth based on application QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). A basic object ob_k has a size δ_k (in bytes) and needs to be downloaded by the processors that use it with frequency f_k. Therefore, these basic object downloads consume an amount of bandwidth equal to $rate_k = \delta_k \times f_k$ on each network link and network card through which the object is communicated. Another thread receives data from the operator's non-leaf children, if any, and performs some computation using downloaded basic objects and/or data received from other operators. The operator produces some output that needs to be passed to its parent operator. The computation of operator n_i (to evaluate the operator once) requires w_i operations, and produces an output of size δ_i.

8.1.2 Platform Model

The target distributed network is a fully connected graph interconnecting a set of resources $\mathcal{R} = \mathcal{P} \cup \mathcal{S}$, where $\mathcal{P}$ denotes compute servers, or *processors* for short, and $\mathcal{S}$ denotes data servers, or *servers* for short. Servers hold and update basic objects, while processors apply operators of the application tree. Each server $S_l \in \mathcal{S}$ (resp. processor $P_u \in \mathcal{P}$) is interconnected to the network via a network card with maximum bandwidth Bs_l (resp. Bp_u). We assume that the same interconnect technology is used to connect all processors, and thus the link between two distinct processors P_u and P_v is bidirectional and has bandwidth bp, while the network link from a server S_l to a processor P_u has bandwidth bs_l; on such links the server sends data and the processor receives it. In addition, each processor $P_u \in \mathcal{P}$ is characterized by a compute speed s_u. We denote the case in which all processors are homogeneous because only one type of CPUs and network cards can be acquired ($Bp_u = Bp$ and $\mathsf{s}_u = \mathsf{s}$) CONSTR-HOM. Correspondingly, we term the case in which the processors are heterogeneous with various compute speeds and network card bandwidth CONSTR-LAN.

Resources operate under the full-overlap, bounded multi-port model [36], where a resource

can be involved in computing, sending data, and receiving data simultaneously. In this model, a resource R_u can be involved in computing, sending data, and receiving data simultaneously. Note that servers only send data, while processors engage in all three activities. A resource R, which is either a server or a processor, can be connected to multiple network links (since we assume a clique network). The "multi-port" assumption states that resource R can send/receive data simultaneously on multiple network links. The "bounded" assumption states that the total transfer rate of data sent/received by resource R is bounded by its network card bandwidth.

8.1.3 Mapping Model and Constraints

Our objective is to purchase/rent a set of processors, and then to map operators, i.e., internal nodes of the application tree, onto these processors. Additionally, if a tree node has at least one leaf child, then it must continuously download up-to-date basic objects from the fixed set of servers, which consumes bandwidth on its processor's network card. Each processor is in charge of one or several operators. For each operator on processor P_u, while P_u computes for the t-th final result, it sends to its parent (if any) the data corresponding to intermediate results for the $(t-1)$-th final result. It also receives data from its non-leaf children (if any) for computing the $(t+1)$-th final result. Recall that all three activities are concurrent. We assume that a basic object can be replicated, in some out-of-band manner specific to the target application (e.g., via a distributed database infrastructure). In this case, a processor can choose among multiple data sources when downloading a basic object. Conversely, if two operators require the same basic object and are mapped to different processors, they must both continuously download that object (and incur the corresponding network overheads).

We denote the mapping of the operators in $\mathcal{N}$ onto the processors in $\mathcal{P}$ using an allocation function a: $a(i) = u$ if operator n_i is assigned to processor P_u. Conversely, $\bar{a}(u)$ is the index set of operators mapped on P_u: $\bar{a}(u) = \{i \mid a(i) = u\}$. We also introduce new notations to describe the location of basic objects. Processor P_u may need to download some basic objects from some servers. We use $DL(u)$ to denote the set of (k,l) couples where processor P_u downloads object ob_k from server S_l. Each processor has to communicate and compute fast enough to achieve the application throughput ρ. A communication occurs only when a child or the parent of a given tree node and this node are mapped on different processors. We have the following constraints:

- Each processor P_u cannot exceed its computation capability:

$$\forall P_u \in \mathcal{P}, \quad \sum_{i \in \bar{a}(u)} \rho \cdot \frac{\mathsf{w}_i}{\mathsf{s}_u} \leq 1 \tag{8.1}$$

- P_u must have enough bandwidth capacity to perform all its basic object downloads and all communication with other processors. The first term corresponds to basic object downloads, the second term corresponds to inter-node communications when a tree node is assigned to P_u and some of its children nodes are assigned to another processor, and the third term corresponds to inter-node communications when a tree node is assigned to P_u and its parent node is assigned to another processor:

$$\forall P_u \in \mathcal{P}, \quad \sum_{(k,l) \in DL(u)} rate_k + \sum_{j \in Child(\bar{a}(u)) \setminus \bar{a}(u)} \rho.\delta_j + \sum_{j \in Parent(\bar{a}(u)) \setminus \bar{a}(u)} \; \sum_{i \in Child(j) \cap \bar{a}(u)} \rho.\delta_i \leq Bp_u \tag{8.2}$$

- Server S_l must have enough bandwidth capacity to support all basic object downloads:

$$\forall S_l \in \mathcal{S}, \quad \sum_{P_u \in \mathcal{P}} \sum_{(k,l) \in DL(u)} rate_k \leq Bs_l \tag{8.3}$$

- The link between server S_l and processor P_u must have enough bandwidth capacity to support all possible object downloads from S_l to P_u:

$$\forall P_u \in \mathcal{P}, \forall S_l \in \mathcal{S}, \quad \sum_{(k,l) \in DL(u)} rate_k \leq bs_{l,u} \tag{8.4}$$

- The link between P_u and P_v must have enough bandwidth capacity to support all possible communications between the nodes mapped on both processors. This constraint can be written similarly to constraint (8.2) above, but without the cost of basic object downloads, and specifying that P_u communicates with P_v:

$$\forall P_u, P_v \in \mathcal{P}$$

$$\sum_{\substack{j \in Child(\bar{a}(u)) \\ \cap \bar{a}(v)}} \rho.\delta_j + \sum_{\substack{j \in Parent(\bar{a}(u)) \\ \cap \bar{a}(v)}} \sum_{\substack{i \in Child(j) \\ \cap \bar{a}(u)}} \rho.\delta_i \leq bp_{u,v} \tag{8.5}$$

8.2 Complexity

Unsurprisingly, most operator mapping problems are NP-hard, because downloading objects with different rates on two identical servers is the same problem as 2-Partition [32]. Let us consider the simplest problem class, i.e., mapping a fully homogeneous left-deep tree application [38] (see Fig. 8.1(b)) without communication costs ($\delta_i = 0$), with objects placed on a fully homogeneous set of servers, onto a fully homogeneous set of processors, LDT-Hom. The objective function consists now in minimizing the number of used processors. It turns out that even this problem is NP-hard, due to the combinatorial space induced by the mapping of basic objects that are shared by several operators.

Definition 8.1. *The problem* LDT-Hom *consists in minimizing the number of processors used in the application execution. K is the prescribed throughput that should not be violated.* LDT-Hom *is the associated decision problem: given a number of processors N, is there a mapping that achieves throughput K?*

Theorem 8.1. LDT-Hom *is NP-complete.*

Proof. First, LDT-Hom belongs to NP. Given an allocation of operators to processors and the download list $DL(u)$ for each processor P_u, we can check in polynomial time that we use no more than N processors, that the throughput of each enrolled processor respects K:

$$K \times |\bar{a}(u)| \frac{\mathsf{w}}{\mathsf{s}} \leq 1 \ ,$$

and that bandwidth constraints are respected.

To establish the completeness, we use a reduction from 3-Partition, which is NP-complete in the strong sense [32]. We consider an arbitrary instance $\mathfrak{I}_1$ of 3-Partition: given $3n$ positive integer numbers $\{a_1, a_2, \ldots, a_{3n}\}$ and a bound R, assuming that $\frac{R}{4} < a_i < \frac{R}{2}$ for all i and that

$\sum_{i=1}^{3n} a_i = nR$, is there a partition of these numbers into n subsets $I_1, I_2, \ldots, I_n$ of sum R? In other words, are there n subsets $I_1, I_2, \ldots, I_n$ such that $I_1 \cup I_2 \ldots \cup I_n = \{1, 2, \ldots, 3n\}$, $I_i \cap I_j = \emptyset$ if $i \neq j$, and $\sum_{j \in I_i} a_j = R$ for all i (and $|I_i| = 3$ for all i). Because 3-Partition is NP-complete in the strong sense, we can encode the $3n$ numbers in unary and assume that the size of $\mathfrak{I}_1$ is $O(n + M)$, where $M = \max_i\{a_i\}$.

We build the following instance $\mathfrak{I}_2$ of LDT-Hom:

- The object set is $\mathcal{OB} = \{ob_1, ..., ob_{3n}\}$, and there are $3n$ servers each holding an object, thus ob_i is available on server S_i. The rate of ob_i is $rate = 1$, and the bandwidth limit of the servers is set to $Bs = 1$.
- The left-deep tree consists of $|\mathcal{N}| = nR$ operators with $\mathsf{w} = 1$. Each object ob_i appears a_i times in the tree (the exact location does not matter), so that there are $|\mathcal{N}|$ leaves in the tree, each associated to a single operator of the tree.
- The platform consists of n processors of speed $\mathsf{s} = 1$ and bandwidth $Bp = 3$. All the link bandwidths interconnecting servers and processors are equal to $bs = bp = 1$.
- Finally we ask whether there exists a solution matching the bounds $1/K = R$ and $N = n$.

The size of $\mathfrak{I}_2$ is clearly polynomial in the size of $\mathfrak{I}_1$, since the size of the tree is bounded by $3nM$. We now show that instance $\mathfrak{I}_1$ has a solution if and only if instance $\mathfrak{I}_2$ does.

Suppose first that $\mathfrak{I}_1$ has a solution. We map all operators corresponding to occurrences of object ob_j, $j \in I_i$, onto processor P_i. Each processor receives three distinct objects, each coming from a different server, hence bandwidths constraints are satisfied. Moreover, the number of operators computed by P_i is equal to $\sum_{j \in I_i} a_i = R$, and the required throughput it achieved because $KR \leq 1$. We have thus built a solution to $\mathfrak{I}_2$.

Suppose now that $\mathfrak{I}_2$ has a solution, i.e., a mapping matching the bound $1/K = R$ with n processors. Due to bandwidth constraints, each of the n processors is assigned at most three distinct objects. Conversely, each object must be assigned to at least one processor and there are $3n$ objects, so each processor is assigned exactly 3 objects in the solution, and no object is sent to two distinct processors. Hence, a processor must compute all operators corresponding to the objects it needs to download, which directly leads to a solution of $\mathfrak{I}_1$ and concludes the proof. ■

Note that this problem becomes polynomial if one adds the additional restriction that no basic object is used by more than one operator in the tree. In this case, one can simply assign operators to $\lceil |\mathcal{N}| \times \mathsf{w}/\mathsf{s} \rceil$ arbitrary processors in a round-robin fashion.

8.2.1 Linear Programming Formulation

In this section, we formulate the Constr optimization problem as an integer linear program (ILP). We deal with the most general instance of the problem Constr-LAN.

Constants

We first define the set of constant values that define our problem. The application tree is defined via parameters par and $leaf$, and the location of objects on servers is defined via parameter obj. Other parameters are defined with the same notations as previously introduced: $\mathsf{comm}_i, \mathsf{w}_i$ for operators, $rate_k$ for object download rates, and Bs_l for server network card bandwidths. More formally:

- $par(i, j)$ is a boolean variable equal to 1 if operator n_i is the parent of n_j in the application tree, and 0 otherwise.
- $leaf(i, k)$ is a boolean variable equal to 1 if operator n_i requires object ob_k for computation, i.e., o_k is a children of n_i in the tree. Otherwise $leaf(i, k) = 0$.
- $obj(k, l)$ is a boolean variable equal to 1 if server S_l holds a copy of object ob_k.
- $\mathsf{comm}_i, \mathsf{w}_i, rate_k, Bs_l$ are rational numbers.

The platform can be built using different types of processors. More formally, we consider a set $\mathcal{C}$ of processor specifications, which we call "classes". We can acquire as many processors of a class $c \in \mathcal{C}$ as needed, although no more than $\mathcal{N}$ processors are necessary overall. We denote the cost of a processor in class c by $cost_c$. Each processor of class c has computing speed s_c and network card bandwidth Bp_c. The link bandwidth between processors is a constant bp, while the link between a server S_l and a processor is bs_l. For each class, processors are numbered from 1 to $|\mathcal{N}|$, and $P_{c,u}$ refers to the u-th processor of class c. Finally, ρ is the throughput that must be achieved by the application:

- $cost_c, \mathsf{s}_c, Bp_c, bp, bs_l$ are rational numbers;
- ρ is a rational number.

Variables

Now that we have defined the constants that define our problem we define unknown variables to be computed:

- $x_{i,c,u}$ is a boolean variable equal to 1 if operator n_i is mapped on $P_{c,u}$, and 0 otherwise. There are $|\mathcal{N}|^2.|\mathcal{C}|$ such variables, where $|\mathcal{C}|$ is the number of different classes of processors.
- $d_{c,u,k,l}$ is a boolean variable equal to 1 if processor $P_{c,u}$ downloads object ob_k from server S_l, and 0 otherwise. The number of such variables is $|\mathcal{C}|.|\mathcal{N}|.|\mathcal{OB}|.|\mathcal{S}|$.
- $y_{i,c,u,i',c',u'}$ is a boolean variable equal to 1 if n_i is mapped on $P_{c,u}$, $n_{i'}$ is mapped on $P_{c',u'}$, and n_i is the parent of $n_{i'}$ in the application tree. There are $|\mathcal{N}|^4.|\mathcal{C}|^2$ such variables.
- $used_{c,u}$ is a boolean variable equal to 1 if processor $P_{c,u}$ is used in the final mapping, i.e., there is at least one operator mapped on this processor, and 0 otherwise. There are $|\mathcal{C}|.|\mathcal{N}|$ such variables.

Constraints

Finally, we must write all constraints involving our constants and variables. In the following, unless stated otherwise, i, i', u and u' span set $\mathcal{N}$; c and c' span set $\mathcal{C}$; k spans set $\mathcal{OB}$; and l spans set $\mathcal{S}$. First we need constraints to guarantee that the allocation of operators to processors is a valid allocation, and that all required downloads of objects are done from a server that holds the corresponding object.

- $\forall i \quad \sum_{c,u} x_{i,c,u} = 1$: each operator is placed on exactly one processor;
- $\forall c, u, k, l \;\; d_{c,u,k,l} \leq obj(k, l)$: object ob_k can be downloaded from S_l only if S_l holds ob_k;

- $\forall c,u,k,l \ d_{c,u,k,l} \leq \sum_i x_{i,c,u}.leaf(i,k)$: if there is no operator assigned to $P_{c,u}$ that requires object k, then $P_{c,u}$ does not need to download object k and $d_{c,u,k,l} = 0$ for all server S_l.

- $\forall i,k,c,u \ \ 1 \geq \sum_l d_{c,u,k,l} \geq x_{i,c,u}.leaf(i,k)$: processor $P_{c,u}$ must download object ob_k from exactly one server if there is an operator n_i mapped on this processor that requires ob_k for computation.

The next set of constraints aim at properly constraining variable $y_{i,c,u,i',c',u'}$. Note that a straightforward definition would be $y_{i,c,u,i',c',u'} = par(i,j).x_{i,c,u}.x_{i',c',u'}$, i.e., a logical conjunction between three conditions. Unfortunately, this definition makes our program non-linear as two of the conditions are variables. Instead, for all i,c,u,i',c',u', we write:

- $y_{i,c,u,i',c',u'} \leq par(i,j)$; $y_{i,c,u,i',c',u'} \leq x_{i,c,u}$; $y_{i,c,u,i',c',u'} \leq x_{i',c',u'}$: y is forced to 0 if one of the conditions does not hold.

- $y_{i,c,u,i',c',u'} \geq par(i,j).\left(x_{i,c,u} + x_{i',c',u'} - 1\right)$: y is forced to be 1 only if the three conditions are true (otherwise the right term is less than or equal to 0).

The following constraints ensure that $used_{c,u}$ is properly defined:

- $\forall c,u \ used_{c,u} \leq \sum_i x_{i,c,u}$: processor $P_{c,u}$ is not used if no operator is mapped on it;

- $\forall c,u,i \ used_{c,u} \geq x_{i,c,u}$: processor $P_{c,u}$ is used if at least one operator n_i is mapped to it.

Finally, we have to ensure that the required throughput is achieved and that the various bandwidth capacities are not exceeded, following equations (1)-(5).

- $\forall c,u \quad \sum_i x_{i,c,u}.\rho\frac{\mathsf{w}_i}{\mathsf{s}_c} \leq 1$: the computation of each processor must be fast enough so that the throughput is at least equal to ρ;

- $\forall c,u \quad \sum_{k,l} d_{c,u,k,l}.rate_k \ + \ \sum_{i,i',(c',u')\neq(c,u)} \ y_{i,c,u,i',c',u'}.\rho.\mathsf{comm}_{i'} +$ $\sum_{i,i',(c',u')\neq(c,u)} y_{i',c',u',i,c,u}.\rho.\mathsf{comm}_i \ \leq \ Bp_c$: bandwidth constraint for the processor network cards;

- $\forall l \quad \sum_{c,u,k} \ d_{c,u,k,l}.rate_k \leq Bs_l$: bandwidth constraint for the server network cards;

- $\forall l,c,u \quad \sum_k d_{c,u,k,l}.rate_k \leq bs_l$: bandwidth constraint for links between servers and processors;

- $\forall c,u,c',u'$ with $(c,u) \neq (c',u') \quad \sum_{i,i'} y_{i,c,u,i',c',u'}.\rho.\mathsf{comm}_{i'} + \sum_{i,i'} y_{i',c',u',i,c,u}.\rho.\mathsf{comm}_i \leq bp$: bandwidth constraint for links between processors.

Objective function

We aim at minimizing the cost of used processors, thus the objective function is

$$\min\left(\sum_{c,u} used_{c,u}.cost_c\right).$$

Altogether, we have provided an integer linear program formulation for the constructive problem.

8.3 Heuristics

In this section we propose several polynomial heuristics to solve the operator-placement problem. The code for all of them is available on the web at http://graal.ens-lyon.fr/~vsonigo/code/query-streaming/. Each heuristic works in two steps: (i) an operator placement heuristic determines the number of processors that should be acquired, and decides which operators are assigned to which processors; (ii) a server selection heuristic decides from which server each processor downloads all needed basic objects.

Operator Placement Heuristics

Note that in most of these heuristics, only the most powerful processors and network cards are acquired. However, these are later replaced by the cheapest ones that still fulfill throughput requirements. This is done just after the server selection step, as a third "downgrade" step, in a view to minimizing cost.

Random – While there are some unassigned operators, the Random heuristic picks one of these unassigned operators randomly, say *op*. It then acquires the cheapest possible processor that is able to handle *op* while achieving the required application throughput. If there is no such processor, then the heuristic considers *op* along with one of its children operators or with its parent operator. This second operator is chosen so that it has the most demanding communication requirements with *op* (in an attempt to reduce communication overhead). If no processor can be acquired that can handle both operators together, then the heuristic fails. If the additional operator had already been assigned to another processor, this last processor is sold back.

Comp-Greedy – The Comp-Greedy heuristic first sorts operators in non-increasing order of w_i, i.e., most computationally demanding operators first. While there are unassigned operators, the heuristic acquires the most expensive processor available and assigns the most computationally demanding unassigned operator to it. If this operator cannot be processed on this processor so that the required throughput is achieved, then the heuristic uses a grouping technique similar to that used by the Random heuristic (i.e., grouping the operator with its child or parent operator with which it has the most demanding communication requirement). If after this step some capacity is left on the processor, then the heuristic tries to assign other operators to it. These operators are picked in non-increasing order of w_i, i.e., trying to first assign to this processor the most computationally demanding operator.

Comm-Greedy – The Comm-Greedy heuristic attempts to group operators to reduce communication costs. It picks the two operators that have the largest communication requirements. These two operators are grouped and assigned to the same processor, thus saving costly communication between both processors. There are three cases to consider: (i) both operators were unassigned, in which case the heuristic simply acquires the cheapest processor that can handle both operators; if no such processor is available then the heuristic acquires the most expensive processor for each operator; (ii) one of the operators was already assigned to a processor, in which case the heuristic attempts to accommodate the other operator as well; if this is not possible then the heuristic acquires the most expensive processor for the other operator; (iii) both operators were already assigned on two different processors, in which case the heuristic attempts

to accommodate both operators on one processor and sell the other processor; if this is not possible then the current operator assignment is not changed.

Subtree-Bottom-Up – This heuristic first acquires as many most expensive processors as there are al-operators and assigns each al-operator to a distinct processor. The heuristic then tries to merge the operators with their father on a single machine, in a bottom-up fashion (possibly returning some processors). Consider a processor on which one or more operators have been assigned. The heuristic first tries to allocate as many parent operators of the currently assigned operators to this processor. If some parent operators cannot be assigned to this processor, then one or more new processors are acquired. This mechanism is used until all operators have been assigned to processors.

Object-Grouping – For each basic object, this heuristic counts how many operators need this basic object. This count is called the "popularity" of the basic object. The al-operators are then sorted by non-increasing sum of the popularities of the basic objects they need. The heuristic starts by acquiring the most expensive processor and assigns to it the first al-operator. The heuristic then attempts to assign to it as many other al-operators that require the same basic objects as the first al-operator, taken in order of non-increasing popularity, and then as many non al-operators as possible. This process is repeated until all operators have been assigned.

Object-Availability – This heuristic takes into account the distribution of basic objects on the servers. For each object k the number av_k of servers handling object o_k is calculated. Al-operators in turn are treated in increasing order of av_k of the basic objects they need to download. The heuristic tries to assign as many al-operators downloading object k as possible on a most expensive processor. The remaining internal operators are assigned similarly to Comp-Greedy, i.e., in decreasing order of w_i of the operators.

Server Selection Heuristics

Once an operator placement heuristic has been applied, each al-operator is mapped on a processor, which needs to download basic objects required by the operator. Thus, we need to specify from which server this download should occur. For the Random heuristic, once the mapping of operators onto processors is fixed, we associate randomly a server to each basic object a processor has to download.

For all other heuristics, we use a more sophisticated heuristic, using three loops. The first loop assigns objects that are held exclusively by a single server. If not all downloads can be guaranteed, the heuristic fails. The second loop associates as many downloads as possible to servers that provide only one basic object type. The last loop finally tries to assign the remaining basic objects that must be downloaded. For this purpose, objects are treated in decreasing order of nbP/nbS, where nbP is the remaining number of processors that need to download the object, and nbS is the number of servers where the object still can be downloaded. In the decision process, servers are considered in decreasing order of the minimum between the remaining bandwidth capacity of the servers network card, and the bandwidth of the communication link.

Once servers have been selected, processors are downgraded if possible: each processor is replaced by a less expensive model that fulfills the CPU and network card requirements of the allocation.

Table 8.1: Incremental costs for increases in processor performance or network card bandwidth relative to a $7,548 base configuration (based on data from the Dell Inc. web site, as of early March 2008).

Processor			Network Card		
Performance (GHz)	Cost ($)	Ratio (GHz/$)	Bandwidth (Gbps)	Cost ($)	Ratio (Gbps/$)
11.72	7,548 + 0	1.55×10^{-3}	1	7,548 + 0	1.32×10^{-4}
19.20	7,548 + 1,550	1.93×10^{-3}	2	7,548 + 399	2.51×10^{-4}
25.60	7,548 + 2,399	2.38×10^{-3}	4	7,548 + 1,197	4.57×10^{-4}
38.40	7,548 + 3,949	3.12×10^{-3}	10	7,548 + 2,800	9.66×10^{-4}
46.88	7,548 + 5,299	3.43×10^{-3}	20	7,548 + 5,999	14.76×10^{-4}

8.4 Simulation Results

Simulation Methodology

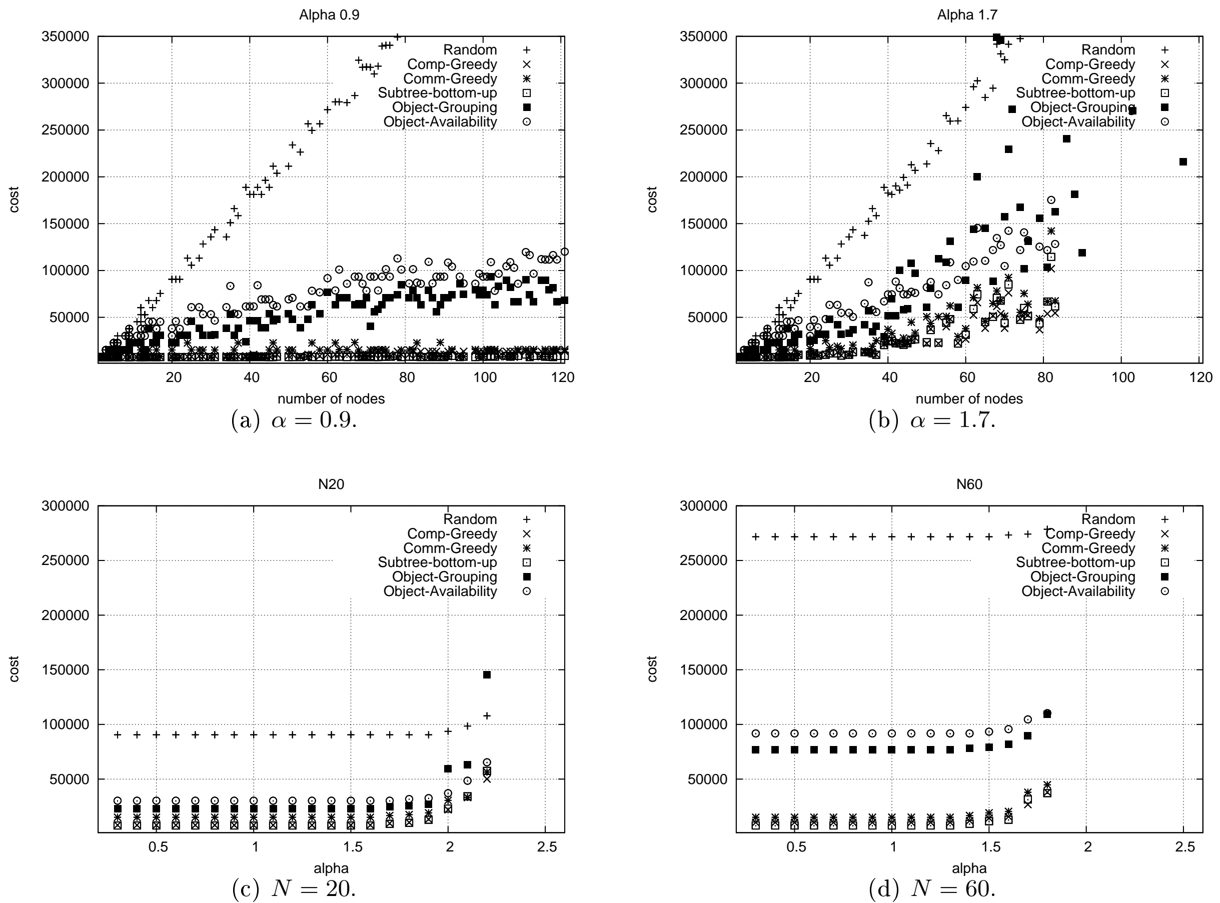
All our simulations use randomly generated binary operator trees with at most N operators, which we vary. All leaves correspond to basic objects, and each basic object is chosen randomly among 15 different types. For each of these 15 basic object types, we randomly choose a fixed size. In simulations with *small object sizes*, in the $\delta_k \in [5, 30]$ MB range, whereas *large object sizes* are in the $\delta_k \in [450, 530]$ MB range. The download frequency for basic objects is either *low* ($f_k = 1/50s$) or *high* ($f_k = 1/2s$). Recall that the download rate for object o_k is then computed as $rate_k = \delta_k \times f_k$.

The computation amount w_i for an operator n_i (a non-leaf node in the tree) depends on its children l and r (basic object or operator): $w_i = (\delta_l + \delta_r)^\alpha$, where α is a constant fixed for each simulation run, and δ is either the size of the basic object, or the amount of data sent by the child operator. The same principle is used for the output size of each operator, setting for all simulations $\delta_i = \delta_l + \delta_r$. The application throughput ρ is fixed to 1 for all simulations. Throughout the whole set of simulations we use the same server architecture: we dispose of 6 servers, each of them equipped with a 10 GB network card. The 15 different types of objects are randomly distributed over the 6 servers. We assume that servers and processors are all interconnected by a 1 GB link. The rest of the platform can be purchased at the costs from Table 8.1 (configurations of Intel's high-end, rack-mountable server, PowerEdge R900).

Results

We present hereafter results for several sets of experiments. The entire set of figures can be found on the web at http://graal.ens-lyon.fr/~vsonigo/code/query-streaming/figures/.

High frequency - small object sizes In the first set of simulations, we study the behavior of the heuristics when the download frequency is high (1/2s) and object sizes small (5-30MB). Figures 8.2(a) and 8.2(b) show the cost as the number of nodes N in the tree varies, with a fixed computation factor α. As expected, Random performs poorly and the platform chosen for an application with around 100 operators or more exceeds a cost of $400,000Figure 8.2(a)), Subtree-bottom-up achieves the best costs, and for an application with 100 operators it finds a platform for the price of $8,745. All Greedy heuristics exhibit similar performance, poorer



(a) $\alpha = 0.9$. (b) $\alpha = 1.7$.

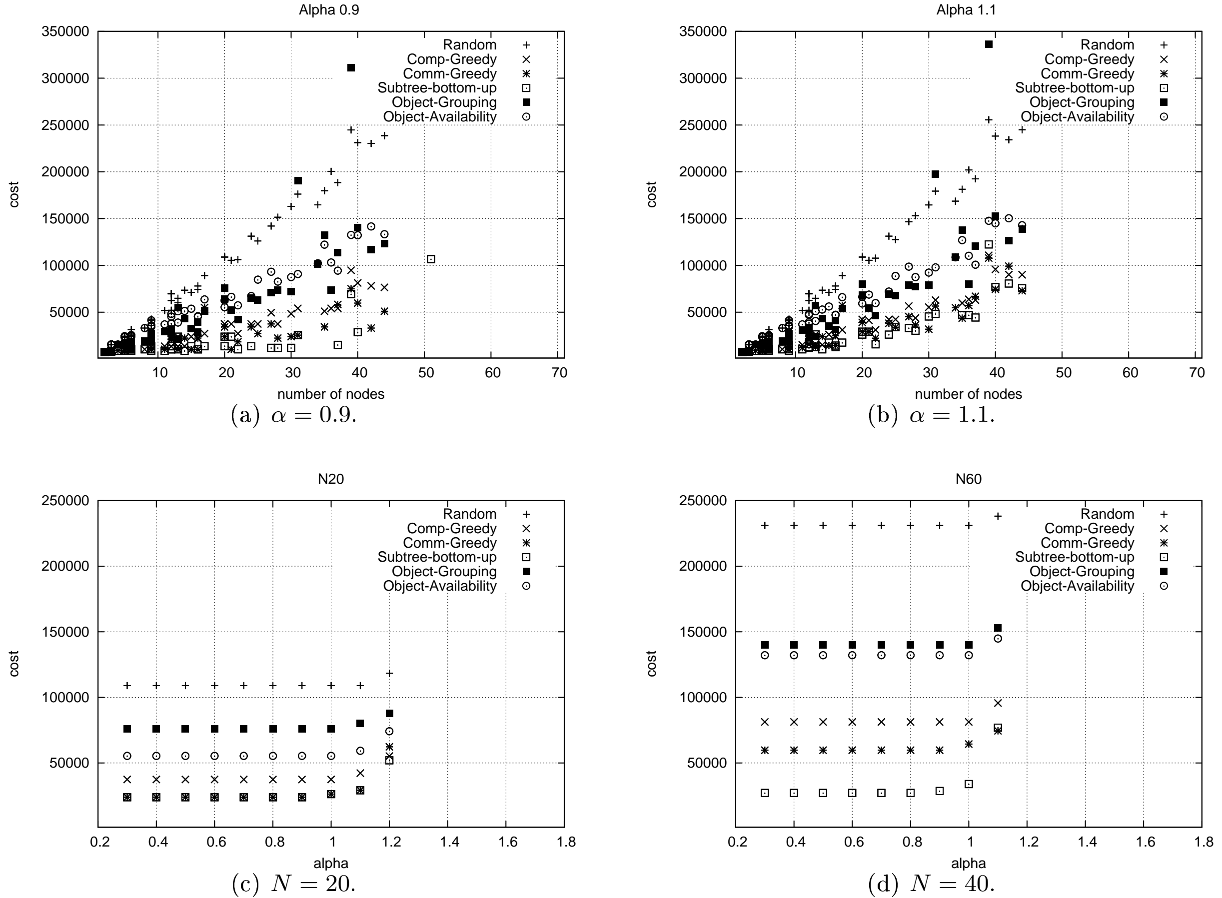
(c) $N = 20$. (d) $N = 60$.

Figure 8.2: Simulation with high frequency and small object sizes.

than Subtree-bottom-up. Perhaps surprisingly, the heuristics that pay special attention to basic objects, Object-Grouping and Object-Availability, perform poorly. With a larger value of α (cf. Figure 8.2(b)) the operator tree size becomes a more limiting factor. For trees with more than 80 operators, almost no feasible mapping can be found. However, the relative performance of our heuristics remains almost the same, with two notable features: a) Object-Grouping still finds some mappings for operator trees with up to 120 operators; b) Comp-Greedy and Object-Greedy perform performs as well as and sometimes better than Subtree-bottom-up when the number of operators increases.

Figure 8.2(d) shows the behavior of the heuristics when N is fixed and the computation factor α increases. Up to a threshold, the α parameter has no influence on the heuristics' performance. When α reaches the threshold, the solution cost of each heuristic increases until α exceeds a second threshold after which solutions can no longer be found. Depending on the number of operators both thresholds have lower or higher values. In the case of small operator trees with only 20 nodes, (see Figure 8.2(c)), the first threshold is for α=1.7 and the second at α=2.2 (vs. α=1.6 and α=1.8 for operator trees of size 60, as seen in Figure 8.2(d)). Subtree-bottom-up behaves in both cases the best, whereas Random performs the poorest. Object-Grouping and Object-Availability change their position in the ranking: for small trees Object-Grouping behaves better, while for larger trees it is outperformed by Object-Availability. The Greedy

heuristics are between Subtree-bottom-up and the object sensitive heuristics. When α is larger, they at times outperform Subtree-bottom-up.



(a) $\alpha = 0.9$. (b) $\alpha = 1.1$.

(c) $N = 20$. (d) $N = 40$.

Figure 8.3: Simulation with high frequency and big object sizes

High frequency - big object sizes With the same experimental setting but large object sizes (450-530MB), the results are similar except that no feasible solution can be found as soon as the trees exceed 45 nodes, Cf. Figures 8.3 (a)-(d). As for small object sizes, we plot two types of figures. Figure 8.3(a) shows results for a fixed α as the number of operators increases. For trees bigger than 45 nodes, almost no feasible solution can be found, both for α smaller than 1 and higher than 1. In general, Subtree-bottom-up still achieves the best costs, but at times it is outperformed by Comm-Greedy. Subtree-bottom-up even fails in two cases (the server selection does not succeed because of bandwidth limitation), while other heuristics find a solution. (N=41 and N=42 in Figure 8.3(a)). The Subtree-bottom-up routine does achieve the best result in terms of processors that have to be purchased. However, in these cases, the server selection heuristic often fails since the operator-processor-mapping fails during the server selection process. (Often the bandwidth of 1 GB between processor and server is not sufficient.) In this experiment Comm-Greedy achieves the best costs among the Greedy heuristics, whereas Random, Object-Availability and Object-Grouping still perform the poorest. When N is fixed, we observe a behavior similar to that for small object sizes. The ranking (Subtree-bottom-up,

Table 8.2: Influence of the download rate on the platform cost, in $, when object sizes are small.

	small object sizes			big object sizes		
N	Comm-Greedy	Obj-Greedy	Subtree-b-up	Comm-Greedy	Obj-Greedy	Subtree-b-up
115	**7947**	13547	8745	**7548**	13547	8745
116	**15495**	13547	**7947**	**15096**	13547	**7548**
117	**7947**	13547	**7947**	**7548**	13547	**7548**
118	**15495**	13547	7548	**15096**	13547	7548
119	**15495**	13547	8745	**15096**	13547	8745

Greedy, object sensitive, and finally Random) remains unchanged (see Figures 8.3(c) and 8.3(d)). When $N = 20$, Comp-Greedy outperforms Object-Greedy and Comm-Greedy finds a feasible solution only once (see Figure 8.3(c)). Object-Availability achieves better results than Object-Grouping. Note that Object-Greedy does not find solutions for experiments with $N = 40$. In the case of $N = 40$ (see Figure 8.3(d)), the ranking is unchanged but for the fact that Object-Availability and Object-Grouping are swapped. Also, in this case, Object-Greedy never succeeds to find a feasible solution, whereas Comm-Greedy achieves the second best results. Note that the failure of Object-Greedy depends on the tree structure, and our results do not mean that Object-Greedy fails for all trees of size higher than 20. In this case again, the solution found by the heuristic for the operator mapping leads to the failure of the server selection process.

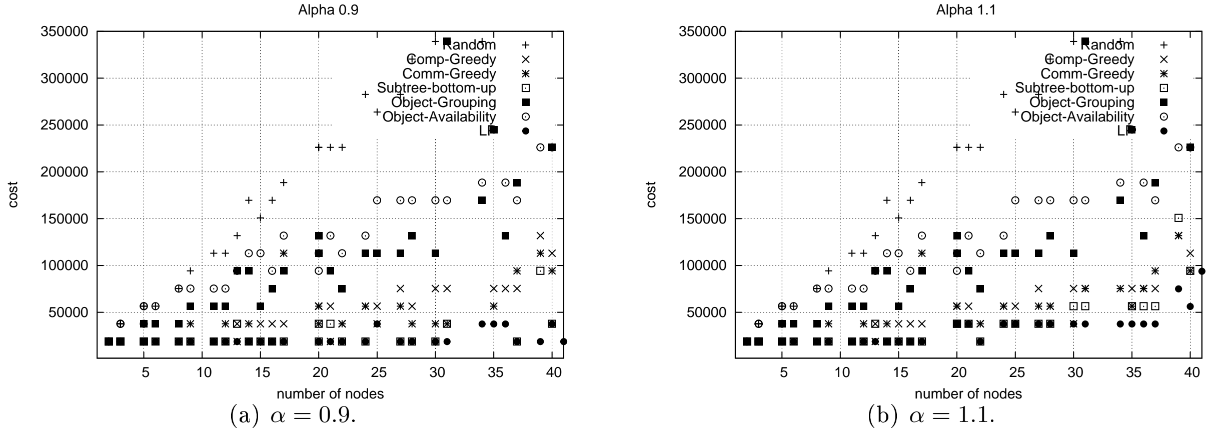
We conclude that a larger number of operators lead earlier to a general failure of the heuristics when α increases.

Low frequency - small object sizes The behaviors of the heuristics with low download frequencies ($f_k = 1/50s$) are almost the same as for high frequency. In general the heuristics lead to the same operator mapping, but in some cases the purchased processors have less powerful network cards. (Cf. Table 8.2).

Low frequency - big object sizes Low frequency slightly improves the success rate of the heuristics. Indeed, because of the lower frequency the links between servers and processors are less congested, and hence the server selection is feasible in more scenarios.

Influence of download rates on the solution In another set of experiments, we study the influence of download rates on the solution. Recall that the download rate of a basic object k is computed by $rate_k = f_k \times \delta_k$. A first result is that frequencies smaller than $1/10s$ have no further influence on the solution. All heuristics find the same solutions for a fixed operator tree.For frequencies between $1/2s$ and $1/10s$, the solution cost changes. In general the cost decreases, but for $N = 160$ the cost for the Object-Grouping heuristic increases. Furthermore, the heuristic ranking remains: Subtree-bottom-up, followed by the Greedy family, followed by the object sensitive ones, and Random. Interestingly, the costs of Object-Availability decrease with the number of operators. In this case the number of operators that need to download a basic object increases, and hence the privileged treatment of basic objects in order of availability on servers becomes more important.

We also tested the importance of the number of basic object replications on the servers. Initially we ran experiments on different server configurations, with basic objects either not



(a) $\alpha = 0.9$. (b) $\alpha = 1.1$.

Figure 8.4: Comparisons of the heuristic and of the optimal LP solution on homogeneous platforms.

replicated or replicated on all servers. However, we did not observe a significant difference in the results across different server configurations. We conclude that the level of replication of basic objects on servers may matter for application trees with specific structures and download frequencies, but that in general we can consider that this parameter has little or no effect on the heuristics' performance.

Comparison of the heuristics to a ILP solution on a homogeneous platform The last set of experiments is dedicated to the evaluation of our heuristics versus a lower bound given by the solution of our ILP. We use the commercial Cplex 11 solver to solve our linear program. Unfortunately, the ILP is so enormous that, even when using only 5 possible groups of processors and using trees with 30 operators, the ILP description file could not be opened in Cplex. For trees with 20 operators, Cplex returns the optimal solution, which consists in all cases in buying a single processor. Therefore, we decided to compare the heuristic solution with the optimal solution only in a homogeneous setting, in which there is only a single processor type. In this case we can skip the downgrading step after the server allocation step. Both for α values lower and higher than 1, Subtree-bottom-up finds the optimal solution in most of the cases (see Figures 8.4(a) and 8.4(b)). The same ranking of the heuristics holds in the homogeneous setting: Subtree-bottom up, the Greedy family, followed by Object-Grouping, Object-Availability and finally Random. Focusing on the Greedy family, we observe that in most cases Comm-Greedy achieves the best cost.

Summary of results

Results show that all our more sophisticated heuristics perform better than the simple random approach. Unfortunately, the object sensitive heuristics, Object-Grouping and Object-Availability, do not show the desired performance. We believe that in some situations these heuristics could lead to good performance, but this is not observed on our set of random application configurations. We have found that Subtree-bottom-up outperforms other heuristics in most situations and also produces results very close to the optimal (for the cases in which we were able to determine the optimal). There are some cases for which Subtree-bottom-up fails. In such cases our results suggest that one should use one of our Greedy heuristics.

Chapter 9

Multiple Concurrent Applications

This chapter investigates the operator mapping problem for in-network stream-processing for multiple concurrent applications, where the goal is to compute some final data at some desired rate. As there are multiple applications, different operator trees may share common subtrees. Therefore, it may be possible to reuse some intermediate results in different application trees.

We consider target platforms that are either fully homogeneous, or with a homogeneous network but heterogeneous servers, or fully heterogeneous. Our specific contributions are twofold: (i) we formalize operator mapping problems for multiple in-network stream-processing applications and give their complexity; (ii) we propose a number of algorithms to solve the problems and evaluate them via extensive simulation experiments. One of the primary objectives of these heuristics is to reuse intermediate results shared by multiple applications. Our quantitative comparisons of these heuristics in simulation demonstrates the importance of choosing appropriate processors for operator mapping.

9.1 Framework

We study operator mapping for multiple trees of operators which should be executed concurrently. The models we use for applications, platform and mapping are quite similar to the ones used in Chapter 8, with the following basic differences: First, the application model deals now with several applications, where each application has its own QoS constraint. Second, in the platform model we do not distinguish between servers and processors anymore. Basic objects are situated at the processors in the platform and these processors can also be used for computation. As the computation platform already exists, we decide about the processors we use for computation. So it may happen, that processors already own the basic objects that the mapped operators need for their computations later on. For the convenience of the reader, we restate the model in detail.

9.1.1 Application Model

We consider $\mathcal{K}$ applications, each needing to perform several operations organized as a binary tree (see Figure 9.1). Operators are taken from the set $\mathcal{OP} = \{op_1, op_2, \ldots\}$, and operations are initially performed on basic objects from the set $\mathcal{OB} = \{ob_1, ob_2, \ldots\}$. These basic objects are made available and continuously updated at given locations in a distributed network. Operators higher in the tree rely on previously computed intermediate results, and they may also require to download basic objects periodically.

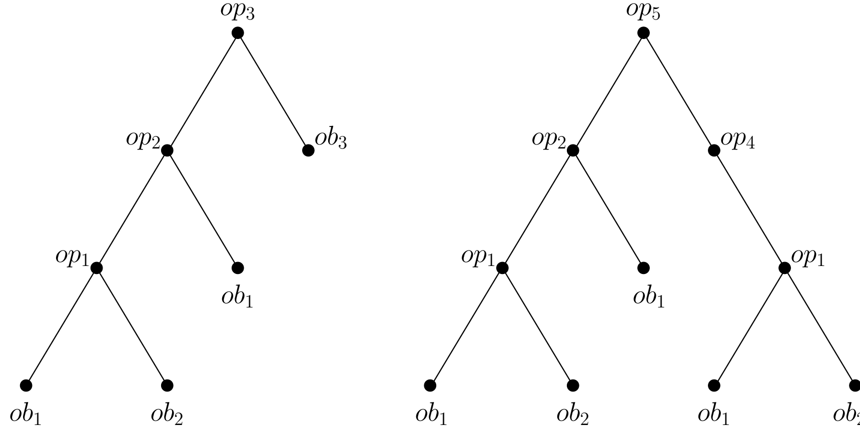


Figure 9.1: Sample applications structured as binary trees of operators.

For an operator op_p we define $objects(p)$ as the index set of the basic objects in $\mathcal{OB}$ that are needed for the computation of op_p, if any; and $operators(p)$ as the index set of operators in $\mathcal{OP}$ whose intermediate results are needed for the computation of op_p, if any. We have the constraint that $|objects(p)| + |operators(p)| \leq 2$ since application trees are binary. An application is fully defined by the operator at the root of its tree. For instance, if we consider Fig. 9.1, we have one application rooted on op_3, and another application rooted on op_5. Operator op_1 needs to download objects ob_1 and ob_2, while operator op_2 downloads only object ob_1 but also requires an intermediate result from operator op_1.

The tree structure of application k is defined with a set of labeled nodes. The i-th internal node in the tree of application k is denoted as $n_i^{(k)}$, its associated operator is denoted as $op(n_i^{(k)})$, and the set of basic objects required by this operator is denoted as $ob(n_i^{(k)})$.

- Node $n_1^{(k)}$ is the root node.
- Let $op_p = op(n_i^{(k)})$ be the operator associated to node $n_i^{(k)}$. Then node $n_i^{(k)}$ has $|operators(p)|$ child nodes, denoted as $n_{2i}^{(k)}, n_{2i+1}^{(k)}$ if they exist.
- Finally, the parent of a node $n_i^{(k)}$, for $i > 1$, is the node of index $\lfloor i/2 \rfloor$ in the same tree.

The applications must be executed so that they produce final results, where each result is generated by executing the whole operator tree once, at a target rate. Each application has its application *throughput*, $\rho^{(k)}$, and the specification of this target throughput is a QoS requirement for each application. Each operator in the tree of the k-th application must compute (intermediate) results at a rate at least as high as the target application throughput $\rho^{(k)}$. We keep the concept that operator op_p executes two concurrent threads in steady-state:

- It periodically downloads the most recent copies of the basic objects in $objects(p)$, if any. Note that these downloads may simply amount to constant streaming of data from sources that generate data streams. Each download has a prescribed cost in terms of bandwidth based on application QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). A basic object ob_j has a size δ_j (in bytes) and needs to be downloaded by the processors that use it for application k with frequency $f_j^{(k)}$. Therefore,

these basic object downloads consume an amount of bandwidth equal to $rate_j^{(k)} = \delta_j \times f_j^{(k)}$ on each network link and network card through which this object is communicated for application k. Note that if a processor requires object ob_j for several applications with different update frequencies, it downloads the object only once at the maximum required frequency $rate_j = \max_k\{rate_j^{(k)}\}$.

- It receives intermediate results computed by $operators(p)$, if any, and it performs some computation using basic objects it is continuously downloading, and/or data received from other operators. The operator produces some output, which is either an intermediate result which will be sent to another operator, or the final result of the application (root operator). The computation of operator op_p (to evaluate the operator once) requires w_p operations, and produces an output of size δ_p.

9.1.2 Platform Model

The target distributed network is a fully connected graph (i.e., a clique) interconnecting a set of processors $\mathcal{P}$. These processors can be assigned operators of the application tree and perform some computation. Some processors also hold and update basic objects. Each processor $P_u \in \mathcal{P}$ is interconnected to the network via a network card with maximum bandwidth Bp_u. The network link between two distinct processors P_u and P_v is bidirectional and has bandwidth $bp_{u,v}(= bp_{v,u})$ shared by communications in both directions. In addition, each processor $P_u \in \mathcal{P}$ is characterized by a compute speed s_u. As in Chapter 8, resources operate under the full-overlap, bounded multi-port model [35]. The case in which some dedicated processors are only providing basic objects but cannot be used for computations is obtained simply by setting their compute speed to 0.

9.1.3 Mapping Model and Constraints

Our objective is to map internal nodes of application trees onto processors. As explained in Section 8.1.1, if the operator associated to a node requires basic objects, the processor in charge of this internal node must continuously download up-to-date basic objects, which consumes bandwidth on its processor's network card. Each used processor is in charge of one or several nodes, and the concurrent activities are almost the same as for single applications:

If there is only one node on processor P_u, while the processor computes for the t-th final result it sends to its parent (if any) the data corresponding to intermediate results for the $(t-1)$-th final result and also receives data from its children (if any) for computing the $(t+1)$-th final result.

Note however that different nodes can be assigned to the same processor. In this case, the same overlap happens, but possibly on different result instances (an operator may be applied for computing the t_1-th result while another is being applied for computing the t_2-th). A particular case is when several nodes with the same operator are assigned to the same processor. In this case, computation is done only once for this operator, but it should occur at the highest required rate among those of the corresponding applications.

Ditto in this mapping model, basic objects can be duplicated, and thus available and updated at multiple processors. In this case, a processor can choose among multiple data sources when downloading a basic object, or perform a local access if the basic object is available locally.

We use a similar allocation function to the one in Chapter 8, a, to denote the mapping of the nodes onto the processors in $\mathcal{P}$: $a(k,i) = u$ if node $n_i^{(k)}$ is mapped to processor P_u. Conversely, $\bar{a}(u)$ is the index set of nodes mapped on P_u: $\bar{a}(u) = \{(k,i) \mid a(k,i) = u\}$. Also, we denote by $a_{op}(u)$ the index set of operators mapped on P_u: $a_{op}(u) = \{p \mid \exists (k,i) \in \bar{a}(u) \;\; op_p = op(n_i^{(k)})\}$. We introduce the following notations:

- $Ch(u) = \{(p,v,k)\}$ is the set of (operator, processor, application) tuples such that processor P_u needs to receive an intermediate result computed by operator op_p, which is mapped to processor P_v, at rate $\rho^{(k)}$; operators op_p are children of $a_{op}(u)$ in the operator tree.
- $Par(u) = \{(p,v,k)\}$ is the set of (operator, processor, application) tuples such that P_u needs to send to P_v an intermediate result computed by operator op_p at rate $\rho^{(k)}$; $p \in a_{op}(u)$ and the sending is done to the parents of op_p in the operator tree.
- $DL(u) = \{(j,v,k)\}$ is the set of (object, processor, application) tuples where P_u downloads object ob_j from processor P_v at rate $\rho^{(k)}$.

The formal definition of $Ch(u)$ and $Par(u)$ is as follows. We first define two sets of tuples, $ACh(u)$ and $APar(u)$, used to account for communications for the same data but for different applications:

$$ACh(u) = \left\{(p,v,k) \mid \exists i, p' \quad p \in a_{op}(v); p' \in a_{op}(u); p \in operators(p'); op_p = op(n_i^{(k)}); op_{p'} = op(n_{\lfloor i/2 \rfloor}^{(k)}\right\}$$

$$APar(u) = \left\{(p,v,k) \mid \exists i, p' \quad p \in a_{op}(v); p' \in a_{op}(u); p \in operators(p'); op_p = op(n_i^{(k)}); op_{p'} = op(n_{\lfloor i/2 \rfloor}^{(k)}\right\}$$

Then we determine which application has the higher throughput for redundant entries, where arg max randomly chooses one application if there are equalities:

$$kchosen(p,v,X) = \arg\max_{k \in \mathcal{K}} \left\{ \;\; \rho^{(k)} \mid \exists (p,v,k) \in X \right\}$$

Finally, $X(u) = \{(p,v,kchosen(p,v,AX)) \mid op_p \in \mathcal{OP}, P_v \in \mathcal{P}\}$. X stands for Ch or Par, and we have thus thus fully defined $Ch(u)$ and $Par(u)$.

Given these notations, we can now express constraints for the application throughput: each processor must compute and communicate fast enough to respect the prescribed throughput of each application which is being processed by it. The computation constraint is expressed below. Note that each operator is computed only once at the maximum required throughput.

$$\forall P_u \in \mathcal{P} \quad \sum_{p \in a_{op}(u)} \left(\max_{(k,i) \in \bar{a}(u) \mid op(n_i^{(k)}) = op_p} \left(\rho^{(k)}\right) \frac{\mathsf{w}_p}{\mathsf{s}_u} \right) \leq 1 \; . \tag{9.1}$$

Communication occurs only when a child or the parent of a given node and this node are mapped on different processors. In other terms, we neglect intra-processor communications. An operator computing for several applications may send/receive results to/from different processors. If the parent/child nodes corresponding to the different applications are mapped onto the same processor, the communication is done only once, at the most constrained throughput. This throughput, as well as the processors with which P_u needs to communicate, are obtained via $Ch(u)$ and $Par(u)$. In these expressions $v \neq u$ since we neglect intra-processor-communications.

P_u must have enough bandwidth capacity to perform all its basic object downloads, to support downloads of the basic objects it may hold, and also to perform all communication

with other processors, all at the required rates. This is expressed in Eq. 9.2. The first term corresponds to basic object downloads; the second term corresponds to download of basic objects from other processors; the third term corresponds to inter-node communications when a node is assigned to P_u and its parent node is assigned to another processor; and the last term corresponds to inter-node communications when a node is assigned to P_u and some of its children nodes are assigned to another processor.

$$\forall P_u \in \mathcal{P} \quad \sum_{(j,v,k)\in DL(u)} rate_j^{(k)} + \sum_{P_v \in \mathcal{P}} \sum_{(j,u,k)\in DL(v)} rate_j^{(k)} + \sum_{(p,v,k)\in Ch(u)} \delta_p \rho^{(k)} + \sum_{(p,v,k)\in Par(u)} \delta_p \rho^{(k)} \le Bp_u \tag{9.2}$$

Finally, we need to express the fact that the link between processor P_u and processor P_v must have enough bandwidth capacity to support all possible communications between the nodes mapped on both processors, as well as the object downloads between these processors. Eq. 9.3 is similar to Eq. 9.2, but it considers two specific processors:

$$\forall P_u, P_v \in \mathcal{P} \quad \sum_{(j,v,k)\in DL(u)} rate_j^{(k)} + \sum_{(j,u,k)\in DL(v)} rate_j^{(k)} + \sum_{(p,v,k)\in Ch(u)} \delta_p \rho^{(k)} + \sum_{(p,v,k)\in Par(u)} \delta_p \rho^{(k)} \le bp_{u,v} \tag{9.3}$$

9.1.4 Optimization Problems

The overall objective of the operator-mapping problem is to ensure that a prescribed throughput per application is achieved while minimizing a cost function. Several relevant problems can be envisioned.

Proc-Nb minimizes the number of processors enrolled for computations (processors that are allocated at least one node);

Proc-Power minimizes the compute capacity and/or the network card capacity of processors enrolled for computations (e.g., a linear function of both criteria);

BW-Sum minimizes the sum of the bandwidth capacities used by the application;

BW-Max minimizes the maximum percentage of bandwidth used on all links (minimizing the impact of the applications on the network for other users).

Different platform types may be considered depending on the heterogeneity of the resources. We consider the case in which the platform is fully homogeneous ($\mathsf{s}_u = \mathsf{s}$, $Bp_u = Bp$ and $bp_{u,v} = bp$), which we term Hom. The heterogeneous case in which network links can have various bandwidths is termed Het.

Each combination of problems and platforms could be envisioned, but we will see that Proc-Power on a Hom platform is actually equivalent to Proc-Nb. Proc-Nb makes more sense in this setting, while Proc-Power is used for Het platforms only. Both types of platforms are considered for the BW-Sum and BW-Max problems.

9.2 Complexity

Problem Proc-Nb is NP-hard in the strong sense. This is true even for a simple case: a Hom platform and a single application ($|\mathcal{K}| = 1$), that is structured as a left-deep tree, in which all

operators take the same amount of time to compute and produce results of size 0, and in which all basic objects have the same size. In fact, this is exactly the problem stated in Chapter 8 for left-deep-trees and we can use the same reduction to 3-partition (see proof of Theorem 8.1). It turns out that the same proof also holds for PROC-POWER on a HOM platform.

The BW-MAX problem is NP-hard because downloading objects with different rates on two processors is the same problem as 2-Partition, which is known to be NP-hard [32]. Here is a sketch of the straightforward proof, which holds even in the case of a single application. Consider an application in which all operators produce zero-size results, and in which each basic object is used only by one operator. Consider three processors, with one of them holding all basic objects but unable to compute any operator. The two remaining processors are able to compute all the operators, and they are connected to the first one with identical network links. Such an instance can be easily constructed. The problem is then to partition the set of operators in two subsets so that the bandwidth consumption on the two network links in use is as equal as possible. This is exactly the 2-Partition problem.

The BW-SUM problem is NP-hard because it can be reduced to the Knapsack problem, which is NP-hard [32]. Here is a proof sketch for a single application. Consider the same application as for the proof of the NP-hardness of BW-MAX above. Consider two identical processors, A and B, with A holding all basic objects. Not all operators can be executed on A and a subset of them need to be executed on B. Such an instance can be easily constructed. The problem is then to determine the subset of operators that should be executed on A. This subset should satisfy the constraint that the computational capacity of A is not exceeded, while maximizing the bandwidth cost of the basic objects associated to the operators in the subset. This is exactly the Knapsack problem.

All these problems can be solved thanks to an integer linear program (see Section 9.3 for the ILP formulations). However, they cannot be solved in polynomial time (unless P=NP). Therefore, in the next section we describe polynomial-time heuristics for one of these problems.

9.3 Linear Programming Formulation

In this section, we give an integer linear program (ILP) formulation of the PROC-POWER-HET, BW-SUM-HET and BW-MAX-HET problems, in terms of an integer linear program (ILP). These are the most general versions of our operator-mapping problems. More restricted versions, e.g., with HOM platforms, can be solved using the same ILPs. We describe the input data to the ILP, its variables, its constraints, and finally its objective functions.

In all that follows, i and i' are indices spanning nodes in set of nodes of an application tree; p and p' are indices spanning operators in $\mathcal{OP}$; j is an index spanning objects in $\mathcal{OB}$; u, u', and v are indices spanning processors in $\mathcal{P}$; k is an application index spanning $\mathcal{K}$.

9.3.1 Input Data

Parameters δ_i, w_i for operators, $rate_j^{(k)}$ for object download rates, and $\mathsf{s}_u, Bp_u, bp_{u,v}$ for processors and network elements, are rational numbers and defined in Section **??**. $\rho^{(k)}$ is a rational number that represents the throughput QoS requirement for application k. For convenience, we also introduce families of boolean parameters: *par*, *oper*, and *object*, that pertain to application trees; and *obj*, that pertain to location of objects on processors. We define these parameters hereafter:

- $par(k, i, i')$ is equal to 1 if internal node $n_i^{(k)}$ is the parent of $n_{i'}^{(k)}$ in the tree of application k, and 0 otherwise.
- $oper(k, i, p)$ is equal to 1 if $op(n_i^{(k)}) = p$, and 0 otherwise.
- $object(k, i, j)$ is equal to 1 if node $n_i^{(k)}$ needs object ob_j (i.e., $p \in objects(op(n_i^{(k)})))$, and 0 otherwise.
- $obj(u, j)$ is equal to 1 if processor P_u owns a copy of object ob_j, and 0 otherwise.

9.3.2 Variables

- $x_{k,i,u}$ is a variable equal to 1 if node $n_i^{(k)}$ is mapped on P_u, and 0 otherwise.
- $d_{j,u,v,k}$ is a variable equal to 1 if processor P_u downloads object ob_j for application k from processor P_v, and 0 otherwise.
- $y_{k,i,u,i',u'}$ is a variable equal to 1 if $n_i^{(k)}$ is mapped on P_u, $n_{i'}^{(k)}$ is mapped on $P_{u'}$, and $n_i^{(k)}$ is the parent of $n_{i'}^{(k)}$ in the application tree.
- $used_u$ is a variable equal to 1 if there is at least one node mapped to processor P_u, and 0 otherwise.
- $xop_{k,p,u}$ is a variable equal to 1 if op_p of application k is mapped to processor P_u, and 0 otherwise.
- $yop_{k,p,u,p',u'}$ is a variable equal to 1 if op_p of application k is mapped on processor P_u, $op_{p'}$ of application k is mapped on processor $P_{u'}$, and op_p is a parent of $op_{p'}$ in application k, and 0 otherwise.
- $Ch_{u,p,v,k}$ is a variable equal to 1 if $(p, v, k) \in Ch(u)$, and 0 otherwise.
- $Par_{u,p,v,k}$ is a variable equal to 1 if $(p, v, k) \in Par(u)$, and 0 otherwise.
- $rho_{u,p}$ is a rational variable equal to the throughput of op_p if it is mapped on processor P_u, and 0 otherwise.
- $ratemax_{j,u,v}$ is a rational variable equal to the download rate of object ob_j by processor P_u from processor P_v, and 0 otherwise.

9.3.3 Constraints

We first give constraints to guarantee that the allocation of nodes to processors is valid, and that each required download is done from a server that holds the relevant object.

- $\forall k, i \quad \sum_u x_{k,i,u} = 1$: each node is placed on exactly one processor;
- $\forall j, u, v, k \quad d_{j,u,v,k} \leq obj(v, j)$: object ob_j can be downloaded from processor P_v only if P_v holds it;

- $\forall i, j, u, k \quad 1 \geq \sum_v d_{j,u,v,k} \geq x_{k,i,u}.object(k,i,j)$: processor P_u must download object ob_j from exactly one processor P_v if there is a node $n_i^{(k)}$ mapped on processor P_u that requires ob_j.

The next two constraints aim at properly defining variables y. Note that a straightforward definition would be $y_{k,i,u,i',u'} = par(k,i,i').x_{k,i,u}.x_{k,i',u'}$, but this leads to a non-linear program. Instead we write, for all k, i, u, i', u':

- $y_{k,i,u,i',u'} \leq par(k,i,i')$; $y_{k,i,u,i',u'} \leq x_{k,i,u}$; $y_{k,i,u,i',u'} \leq x_{k,i',u'}$: $y_{k,i,u,i',u'}$ is forced to be 0 if one of these three conditions does not hold.
- $y_{k,i,u,i',u'} \geq par(k,i,j).\left(x_{k,i,u} + x_{k,i',u'} - 1\right)$: $y_{k,i,u,i',u'}$ is forced to be 1 only if the three conditions are true (otherwise the right term is lower than or equal to 0).

The following two constraints ensure that $used_u$ is properly defined:

- $\forall u \; used_u \leq \sum_{k,i} x_{k,i,u}$: processor P_u is not used if no node is mapped to it;
- $\forall k, i, u \; used_u \geq x_{k,i,u}$: processor P_u is used if at least one node n_i is mapped to it.

The following four constraints ensure that $xop_{k,p,u}$ and $yop_{k,p,u}$ are properly defined:

- $\forall i, k, p, u \; xop_{k,p,u} \geq x_{k,i,u}.oper(k,i,p)$: xop is forced to be 1 if operator op_p of application k is mapped on processor P_u;
- $\forall k, p, u \quad xop_{k,p,u} \leq \sum_i x_{k,i,u}.oper(k,i,p)$: xop is forced to be 0 if operator op_p of application k is not mapped on processor P_u;
- $\forall k, p, p', u, u', i, i' \; yop_{k,p,u,p',u'} \leq xop_{k,p,u}$; $yop_{k,p,u,p',u'} \leq xop_{k,p',u'}$; $yop_{k,p,u,p',u'} \leq par(k,i,i')$; $yop_{k,p,u,p',u'} \leq oper(k,i,p)$; $yop_{k,p,u,p',u'} \leq oper(k,i',p')$: $yop_{k,p,u,p',u'}$ is forced to be 0 if one of these conditions does not hold;
- $\forall k, p, p', u, u', i, i' \; yop_{k,p,u,p',u'} \geq par(k,i,i').oper(k,i,p).oper(k.i',p').(xop_{k,p,u}+xop_{k,p',u'}-1)$: $yop_{k,p,u,p',u'}$ is forced to be 1 only if all five conditions are true.

The next four constraints ensure that $Ch_{u,p,v,k}$ and $Par_{u,p,v,k}$ are defined properly:

- $\forall u, p, v, k \quad Ch_{u,p,v,k} \leq \sum_{p'} yop_{k,p',u,p,v}$: in application k, if the parent operator of operator op_p, which is mapped on P_v, is not mapped on processor P_u, Ch is forced to be 0;
- $\forall p', u, p, v, k \quad Ch_{u,p,v,k} \geq yop_{k,p',u,p,v}$: in application k, if operator op_p of application k is mapped to P_v and its parent operator in the application tree is mapped to P_u, Ch is forced to be 1.
- $\forall u, p, v, k \quad Par_{u,p,v,k} \leq \sum_{p'} yop_{k,p',v,p,u}$: in application k, if the parent operator of operator op_p, which is mapped to P_u, is not mapped to processor P_v, Par is forced to be 0;
- $\forall p', u, p, v, k \quad Par_{u,p,v,k} \geq yop_{k,p',v,p,u}$: in application k, if operator op_p is mapped to P_u and its parent operator in the application tree is mapped to P_v, Par is forced to be 1.

The following two constraints ensure that the throughput QoS requirement of each application, $\rho^{(k)}$, is met:

- $\forall k, u, p \;\; rho_{u,p} \geq xop_{k,p,u}.\rho^{(k)}$: the throughput of processor P_u, to which operator op_p of application k is mapped, has to satisfy the throughput QoS requirement of application k;
- $\forall k, p, u, v \;\; ratemax_{p,u,v} \geq d_{p,u,v,k}.rate_p^{(k)}$: the update rate of operator op_p on processor P_u has to satisfy the throughput QoS requirement of application k;

The following constraint ensures that the compute capacity of each processor is not exceeded while meeting QoS throughput requirements:

- $\forall u \;\; \sum_p rho_{u,p} \frac{\mathsf{w}_p}{\mathsf{s}_u} \leq 1$.

The following two constraints ensure that the bandwidth capacity of network elements are not exceeded:

- Bandwidth constraint for the processor network cards:

$$\begin{aligned}\forall u \quad & \sum_{p,v,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,v,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ & \sum_{j,v,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,v,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq Bp_u\end{aligned} \tag{9.4}$$

- Bandwidth constraints for links between processors:

$$\begin{aligned}\forall u, v \quad & \sum_{p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ & \sum_{j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq bp_{u,v}\end{aligned} \tag{9.5}$$

9.3.4 Objective Function

We have to define the objective function to optimize. We have a different definition for each problem:

Proc-Power-Het:

$$\min \left(\sum_{u,p} rho_{u,p} \frac{\mathsf{w}_p}{\mathsf{s}_u} \right) . \tag{9.6}$$

BW-Sum-Het:

$$\begin{aligned}\min & \sum_{u,v,p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{u,v,p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ & \sum_{u,v,j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{u,v,j,k} d_{j,v,u,k}.ratemax_{j,v,u} \; .\end{aligned} \tag{9.7}$$

BW-Max-Het: For this problem we need to add one variable, $bwmax$, and $|\mathcal{P}|^2$ constraints:

$$\begin{aligned}\forall u, v \quad & \sum_{p,k} Ch_{u,p,v,k}.rho_{u,p}.\delta_p + \sum_{p,k} Par_{u,p,v,k}.rho_{u,p}.\delta_p + \\ & \sum_{j,k} d_{j,u,v,k}.ratemax_{j,u,v} + \sum_{j,k} d_{j,v,u,k}.ratemax_{j,v,u} \leq bwmax\end{aligned} \tag{9.8}$$

and the objective becomes: $\min (bwmax)$.

9.4 Heuristics

In this section we propose several polynomial heuristics[1] for the PROC-POWER problem, in which we consider only the compute capacity of processors enrolled for computation. Two heuristics use a random approach to process application nodes, while the others are based on tree traversals. As for the choice of an appropriate resource for the current node, four different processor selection strategies are implemented (and shared by all heuristics). Two selection strategies are *blocking* and two are *non-blocking*. *Blocking* means that once chosen for a given operator op_1, a processor cannot be reused later for another operator op_2, and it is only possible to add relatives (i.e., father or children) of op_1 to this processor. On the contrary, *non-blocking* strategies impose no such restrictions. We start with a description of the four processor selection strategies, and then we move to a brief overview of each heuristic.

Processor Allocation Strategies

(1) Fastest processor first (blocking) – Every time we have to chose a processor, the fastest remaining (not already chosen) processor is chosen.

(2) Biggest network card first (blocking) – Every time we have to chose a processor, the remaining processor with the biggest network card is chosen.

(3) Fastest remaining processor (non-blocking) – The actual amount of computation is subtracted from the computation capability, and the processor with the most remaining computation power is chosen.

(4) Biggest remaining network card (non-blocking) – In this strategy the current (already assigned) communication volume is subtracted from the network card capacity to evaluate the processor whose remaining communication capacity is the biggest. This processor is chosen.

Significance of Node Reuse

Our heuristics, except RandomNoReuse (H1), are designed for node reuse. This means that we try to benefit from the fact that different applications may have common subtrees, i.e., subtrees composed of the same operators. Instead of recomputing the result for such a subtree, we aim at reusing the result. For this purpose we try to add additional communications as can be seen in Figure 9.2. The processor that computes the left op_1 in application 1 sends its result not only to the processor that computes op_2, but also to the processor that computes op_4. The operator op_1 on the right of application 1 no longer has to be computed. In the same way, we save the whole computation of the subtree rooted by op_2 in application 2 when we add the communication between op_2 in application 1 and op_3 in application 2.

We give hereafter a brief overview of each heuristic:

[1] To ensure the reproducibility of our results, the code for all heuristics is available on the web: http://graal.ens-lyon.fr/~vsonigo/code/query-multiapp/.

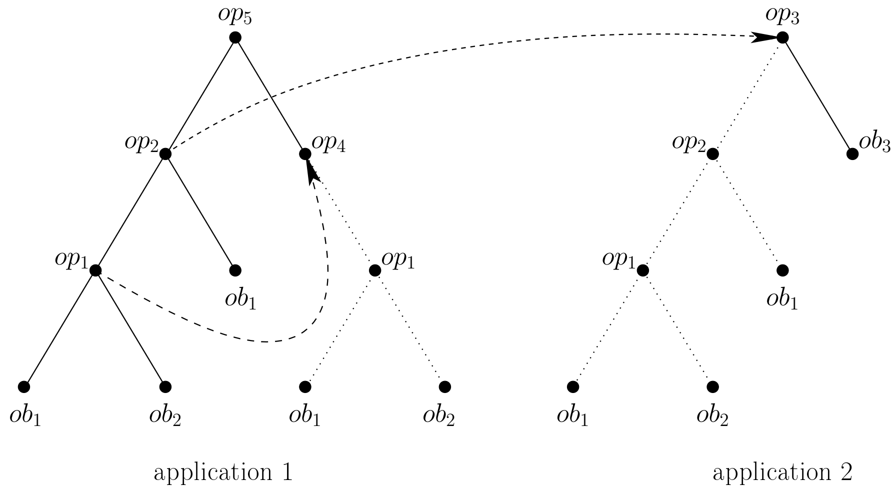


Figure 9.2: Example for the reuse of nodes. op_1 is only computed once and its result is reused for the computation of op_2 and op_4. op_3 uses the result of op_2 in application 1 for its computation.

H1: RandomNoReuse

The **RandomNoReuse** heuristic does not reuse any result. While there are unassigned operators, **RandomNoReuse** randomly picks one of them. If the father is already mapped, it tries to map the operator on the father's processor, or it tries the children's processors, if those are already mapped. If none of these mappings is possible, **RandomNoReuse** chooses a new processor according to the processor selection strategy, and maps the operator. If this is not possible, **RandomNoReuse** fails.

H2: Random

The **Random** random heuristic is more sophisticated as it tries to reuse common results. If the randomly chosen operator has not already been mapped, possibly for another application, we use the same mechanism as in **RandomNoReuse**: first try to map the operator on its father's processor or one of the children's, and in case of failure choose a new processor. But, if the operator has already been mapped somewhere else in the forest, we try to add a link from the already mapped operator to the father of the actual operator to reuse the common result. When this is possible, we mark the whole subtree (rooted at the operator) as mapped. Otherwise, we choose a new processor.

H3: TopDownBFS

The **TopDownBFS** heuristic performs a breadth-first-search (BFS) traversal of all applications. We use an artificial root node to link all applications, i.e., all application roots become children of the artificial root. For each operator, we check whether the operator has not been mapped yet and whether its father has. In this case, **TopDownBFS** tries to map the operator on the same processor as its father, and in case of success continues the BFS traversal. In the case where the actual operator has already been mapped onto one or more processors, **TopDownBFS** tries to add a communication link between the mapped operator and the father of the actual operator: the mapped operator sends its result not only to its father but also to the father of

the actual operator. If none of these two conditions holds, or if the mapping was not possible, **TopDownBFS** tries to map the operator onto a new processor. The processor is chosen according to the processor selection strategy. When the mapping is successful, the BFS traversal is continued, otherwise **TopDownBFS** fails.

H4: TopDownDFS

The **TopDownDFS** heuristic uses the same mechanism as **TopDownBFS**, but operators are treated in depth-first-search (DFS) manner. Thus, each time a mapping of a node is successful, the heuristic continues the DFS traversal of the current application tree.

H5: BottomUpBFS

As the **TopDownDFS** heuristic, the **BottomUpBFS** heuristic makes a BFS traversal of the application forest. For this purpose we use the same mechanism of a new artificial root that links all applications. For each operator, **BottomUpBFS** verifies whether it has already been mapped on a processor. In this case a communication link is added (if possible), connecting the mapped operator and the father of the unmapped operator. If the operator is not yet mapped and if it has some children, we try to map the operator to one of its children's processors. If no such possibility is successful, or if the operator is at the bottom of a tree, **BottomUpBFS** tries to map the operator onto a new processor (where the processor is chosen according to the processor selection strategy). When the mapping is successful, the BFS traversal continues, otherwise **BottomUpBFS** fails.

H6: BottomUpDFS

The **BottomUpDFS** heuristic is similar to **BottomUpBFS**, but instead of a BFS traversal, it performs a DFS traversal of the application forest. This makes the heuristic a little bit more complicated, as there are more cases to be considered. For each node we check if its operator has already been mapped on a processor, and none of its children are. In this case we go up in the tree until we reach the last node n_1 such that there exists a node n_2 somewhere else in the forest which is already mapped, and such that $op(n_1) = op(n_2)$. In this case we try to add a communication between n_2 and the father of n_1 to benefit from the calculated result. If the children have already been mapped we simply try to map the operator to one of the children's processors. If this is not possible or if the additional communication was not possible or again if the operator has not been mapped anywhere in the forest, **BottomUpDFS** tries to map the operator onto a new processor, according to the processor selection strategy. Otherwise **BottomUpDFS** fails.

9.5 Experimental Results

We have conducted several experiments to assess the performance of the different heuristics described in Section 9.4. In particular, we are interested in the impact of node reuse on the number of solutions found by the heuristics.

9.5.1 Experimental Plan

Except for Experiment 1, all application trees are fixed to a size of at most 50 operators, and except for Experiment 5, we consider 5 concurrent applications. The leaves in the tree correspond to basic objects, and each basic object is chosen randomly among 10 different types. The size δ of each object type is also chosen randomly and varies between 3MB and 13MB. The download frequencies of objects for each application, f, as well as the application throughput, ρ, are chosen randomly such that $0 < f \leq 1$ and $1 \leq \rho \leq 2$. The parameters for operators are also chosen randomly. In all experiments (except Experiment 4), the computation amount w_i for an operator lies between 0.5MFlop/sec and 1.5MFlop/sec, and the output size of each operator δ_i is randomly chosen between 0.5MB and 1.5MB.

Throughout most of our experiences we use the following platform configuration (variants will be mentioned explicitly when needed.) We dispose of 30 processors. Each processor is equipped with a network card, whose bandwidth limitation varies between 50MB and 180MB. We use the same range for computation power, i.e., CPU speeds of 50MIPS to 180MIPS. The different processors are interconnected via heterogeneous communication links, whose bandwidth are between 60MB/s and 100MB/s. The 10 different types of objects are randomly distributed over the processors. Execution time and communication time are scaled units, thus execution time is the ratio between computation amount and processor speed, while communication time is the ratio between object size (or output size) and link bandwidth.

To assess performances, we study the relative performance of each heuristic compared to the best solution found by any heuristic. This allows to compare the cost, in amount of resources used, of the different heuristics. The relative performance for the heuristic h is obtained by: $\frac{1}{|runs|}\sum_{r=1}^{|runs|} a_h(r)$, where $a_h(r) = 0$ if heuristic h fails in run r and $a_h(r) = \frac{cost_{best}(r)}{cost_h(r)}$. $cost_{best}(r)$ is the best solution cost returned by one of the heuristics for run r, and $cost_h(r)$ is the cost involved by the solution proposed by heuristic h. Note that in the definition of the relative performance we do account for the case when a heuristic fails on a given instance. The number of runs is fixed to 50 in all experiments. [2]

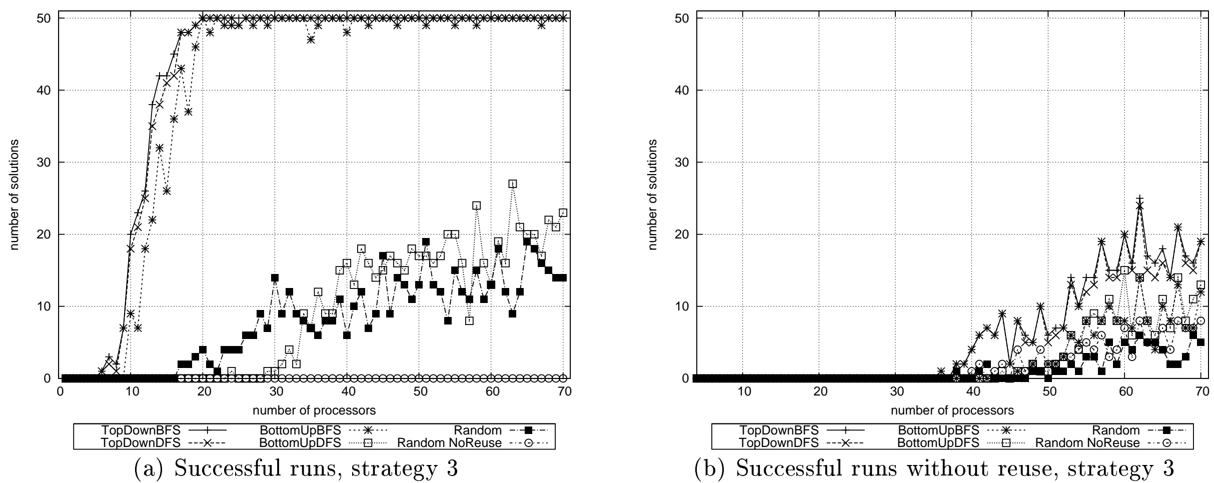
9.5.2 Results

Experiment 1: Number of Processors

In a first set of experiments, we test the influence of the number of available processors, varying it from 1 to 70. Figure 9.3(a) shows the number of successes of the different heuristics using selection strategy 3 (biggest remaining network card). Between 1 and 20 processors, the number of solutions steeply increases for TopDownDFS, TopDownBFS and BottomUpBFS and for higher numbers of processors all three heuristics find solutions for most of the 50 runs. BottomUpDFS finds solutions when more than 30 processors are available. Random already finds solutions when only 20 processors are available, but for the runs with more than 30 processors, it finds fewer solutions than BottomUpDFS. RandomNoReuse is not successful at all, it does not find any solution. To summarize, TopDownBFS finds the most solutions, shortly followed by TopDownDFS and BottomUpBFS. Comparing the success rates of the different selection strategies, all heuristics find the most solutions using strategy 3, followed by strategy 4, strategy 2, and finally strategy 1. But the differences are small. More interesting is the relative performance of the heuristics using the different processor selection strategies in comparison to the number

[2] The complete set of figures summarizing all experimental results is available on the web at http://graal.ens-lyon.fr/~vsonigo/code/query-multiapp/diagrams/.

of solutions. Figure 9.4(a) shows the relative performance using strategy 3. Comparing with Figure 9.4(b), we can conclude that for the same number of successful runs, the performances of the heuristics significantly differ according to the selected processor selection strategy. Using strategy 3 (and also strategies 2 and 4), TopDownDFS performs better than TopDownBFS, which performs better than BottomUpBFS. However, BottomUpBFS outperforms both TopDown heuristics when strategy 1 is used. The performance of BottomUpDFS and of the random ones mirrors exactly the number of successful runs. As for the heuristics without reuse of common subtrees, we see that they do not find results until at least 35 processors are available (strategy 3) or even 60 (strategy 2). Independently of the processor selection strategy, both TopDown heuristics outperform all other heuristics in success and performance, but the results are poor (see Figure 9.3(b)).



(a) Successful runs, strategy 3

(b) Successful runs without reuse, strategy 3

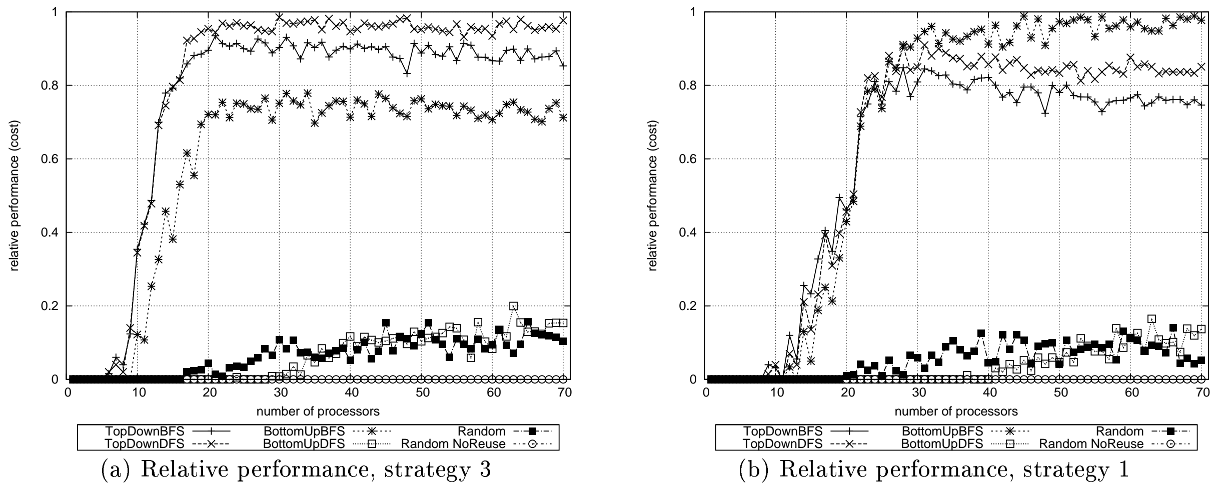
Figure 9.3: Experiment 1: Increasing number of processors. Number of successful runs.

Experiment 2: Number of Applications

In this set of experiments we vary the number of applications, $\mathcal{K}$. As the number of application increases, all heuristics are less successful with strategies 1 and 2 than with strategies 3 and 4, and relative performance is poorer as well. Regardless of the strategy used, both TopDown heuristics show a better relative performance than BottomUpBFS, with the only exception using strategy 1 with a small number of applications (Figure 9.5(a)). BottomUpDFS and both random heuristics perform poorly. For instance, BottomUpDFS only finds solutions with up to 4 applications. The best strategy seems to be strategy 3 in combination with TopDownBFS for more than 10 applications and TopDownDFS for less than 10 applications (see Figure 9.5(b)).

Experiment 3: Application Size

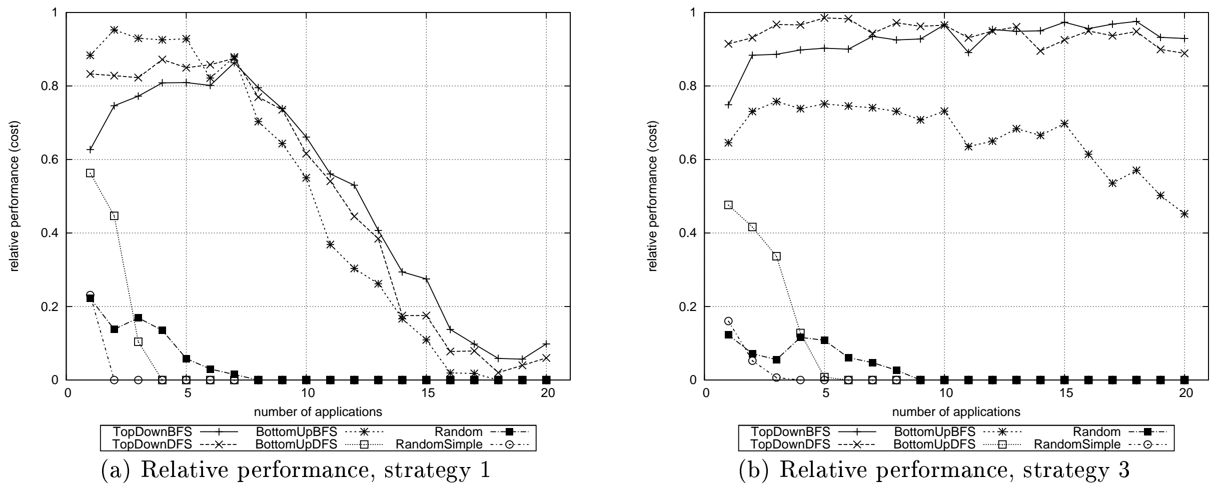
When increasing the application sizes, strategy 3 is the most robust. Up to application sizes of 40 operators, the other strategies are competitive, but for applications bigger than 40 operators both TopDown heuristics and BottomUpBFS achieve the best relative performance and find the most solutions. The success ranking of the three heuristics is the same, independently of the strategy: TopDownBFS finds more solutions than TopDownDFS, which, in turn, finds



(a) Relative performance, strategy 3

(b) Relative performance, strategy 1

Figure 9.4: Experiment 1: Increasing number of processors. Relative performance.



(a) Relative performance, strategy 1

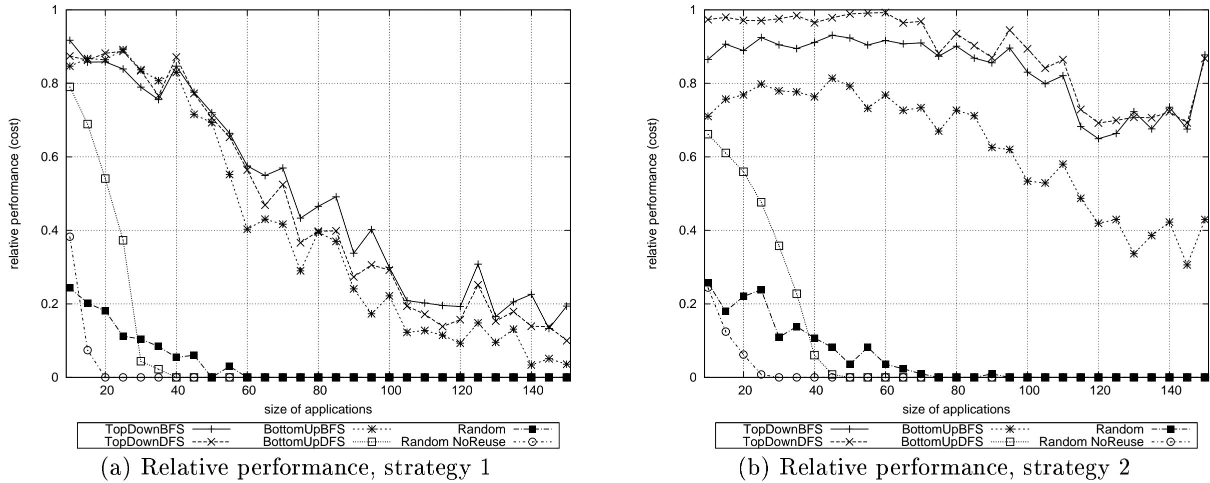
(b) Relative performance, strategy 3

Figure 9.5: Experiment 2: Increasing number of applications.

more solutions than BottomUpBFS. RandomNoReuse finds solutions for applications with fewer than 20 operators, BottomUpBFS up to 40 operators and Random up to 50 operators, but the number of solutions from the latter is poor. As far as relative performance is concerned, both TopDown heuristics achieve the best results for application sizes bigger than 20 using strategy 3. BottomUpDFS is competitive when using strategy 1 for applications smaller than 40 operators (compare Figures 9.6(a) and 9.6(b)). As for the heuristics without reuse of common subtrees, they no longer find results when application sizes exceed 40 operators. TopDown heuristics perform better, and the best strategy is one of the two non-blocking ones (3 or 4).

Experiment 4: Communication-to-Computation Ratio (CCR)

For this experimental set we introduce a new parameter, the CCR, which is the ratio between the mean amount of communications and the mean amount of computations, where the com-



(a) Relative performance, strategy 1

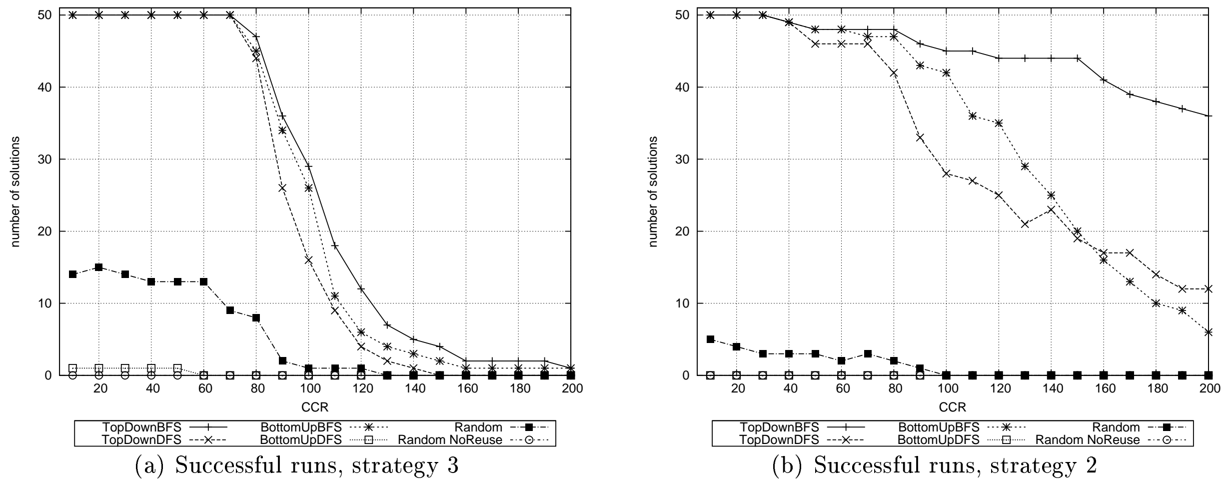
(b) Relative performance, strategy 2

Figure 9.6: Experiment 3: Relative performance for increasing application sizes.

munications correspond to the output sizes of operators (δ_i) and the computations to the computational volume w_i of the operators. When increasing the CCR, strategies 3 and 4 react very sensitively. As can be seen in Figure 9.7(a), TopDownBFS, TopDownDFS and BottomUpBFS have a 100% success rate for CCR ≤ 60, but then the success decreases drastically until no solution is found at all for a CCR of 180 (using strategy 2, TopDownBFS still finds 32 solutions). BottomUpDFS is largely outperformed by Random, and RandomNoReuse fails completely. In this experiment, strategy 2 seems to be the most successful processor selection strategy (see Figure 9.7(b)). TopDownBFS achieves the best results, followed by BottomUpBFS for CCR < 120, and by TopDownDFS for CCR > 120. Interestingly, the relative performances of the heuristics using the different strategies do not directly mirror their success rates. Compare Figures 9.8(a) and 9.8(b): BottomUpBFS finds fewer solutions using strategy 1 than 2, but its relative performance using strategy 1 and CCR smaller than 80 is better than when using strategy 2. Furthermore, TopDownBFS using strategy 1 always finds the most solutions of all heuristics, but its relative performance is only the best when the CCR becomes bigger than 120. Also, TopDownDFS finds fewer solutions than TopDownBFS and BottomUpBFS using strategy 2 and CCR$=30$, but its relative performance is the best.

Experiment 5: Similarity of Applications

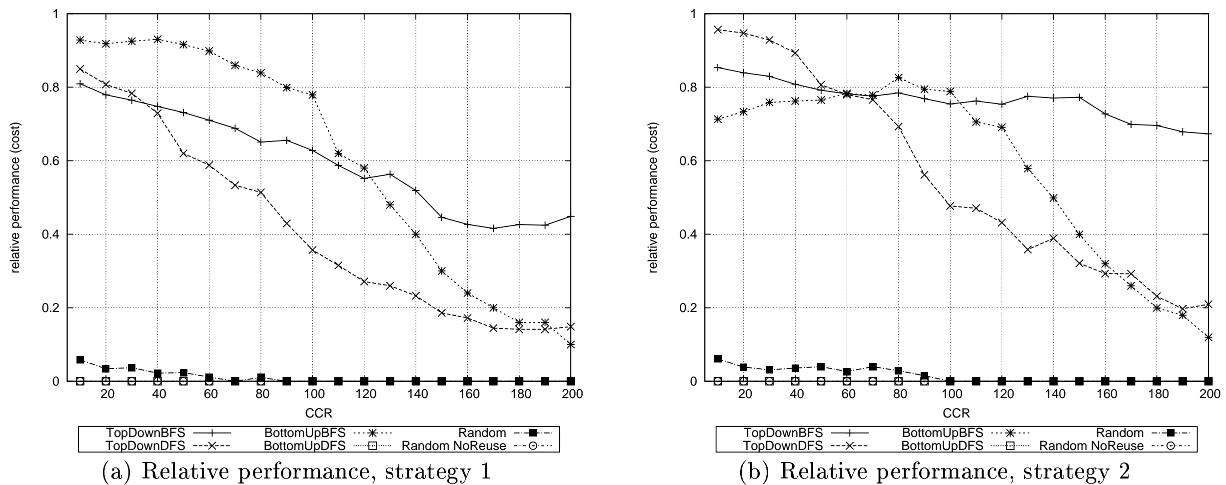
In this last experiment, we use only two applications for each run and the processing platform is smaller, consisting of only 10 processors. We study the influence on our heuristics when applications are very similar or completely different. For this purpose we create applications that differ in more and more operators. Strategies 1 and 2 are more sensitive to application differences and we observe the following ranking for the success of the heuristics: strategy 3 $>$ strategy 4 $>$ strategies 1 and 2, which have similar success rates (compare Figures 9.9(a) and 9.9(b).) The ranking of the heuristics within the different strategies is the same: TopDownBFS is the most successful, followed by TopDownDFS and BottomUpBFS. BottomUpDFS and Random keep the fourth place, while RandomNoReuse fails. TopDownBFS has the best relative performance using the blocking strategies, whereas in the non-blocking cases TopDownDFS achieves the best results, which is important as its success rate is slightly poorer. BottomUpBFS always ranks at



(a) Successful runs, strategy 3

(b) Successful runs, strategy 2

Figure 9.7: Experiment 4: Communication-Computation Ratio CCR. Number of successful runs.



(a) Relative performance, strategy 1

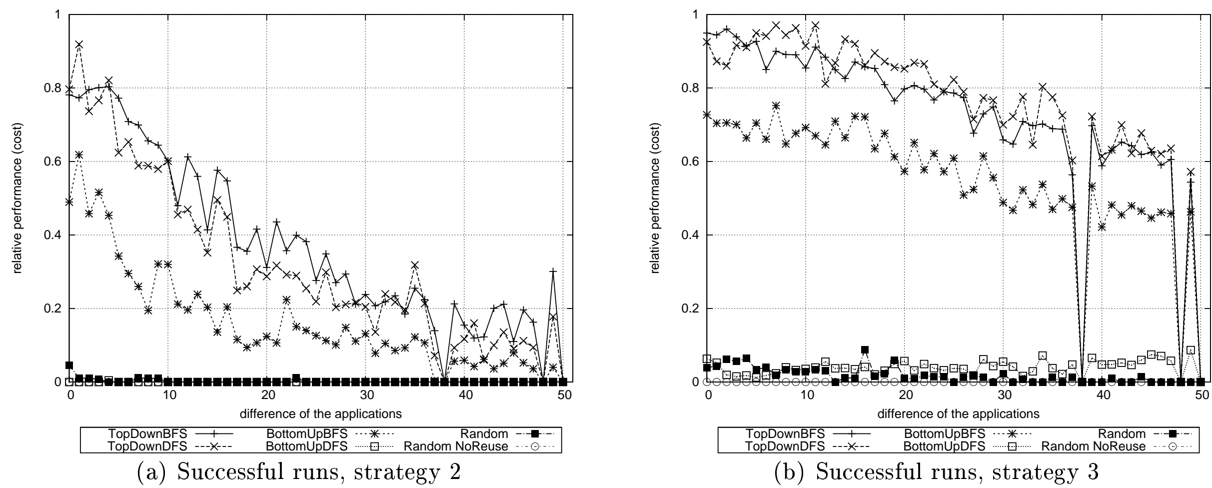
(b) Relative performance, strategy 2

Figure 9.8: Experiment 4: Communication-Computation Ratio CCR. Relative performance.

the third position.

Summary of Experiments

Our results show that a random approach for multiple applications is not feasible. Neglecting the possibility to reuse results from common subtrees dramatically limits the success rate and also the quality of the solution in terms of cost. The TopDown approach turns out to be the best, whereupon in most cases BFS traversal achieves the best result. The BottomUp approach is only competitive using a BFS traversal. The DFS traversal seems unable to reuse results efficiently (it often finds itself with no bandwidth left to perform necessary communications.) Furthermore we see a strong dependency of the processor selection strategy on solution quality. The blocking strategies outperform the non-blocking strategies when the CCR is large. In the other cases, TopDownBFS in combination with strategy 3 proves to be a solid combination.



(a) Successful runs, strategy 2

(b) Successful runs, strategy 3

Figure 9.9: Experiment 5: Similarity of applications.

Conclusion and Perspectives

10.1 Conclusion

In this thesis we have studied multi-criteria optimization problems for large-scale heterogeneous platforms, focusing on streaming applications. Within this application field, we chose three particular application classes and investigated them from a theoretical and practical approach. In the following we detail our contributions.

10.1.1 Replica Placement

In the first part of this thesis, we have introduced and extensively analyzed two important new policies for the replica placement problem. The *Upwards* and *Multiple* policies are natural variants of the standard *Closest* approach, and it may seem surprising that they have not already been considered in the published literature.

On the theoretical side, we have fully assessed the complexity of the *Closest*, *Upwards* and *Multiple* policies, both for homogeneous and heterogeneous platforms. Not surprisingly, all three policies turn out to be NP-complete for heterogeneous nodes, which provides yet another example of the additional difficulties induced by resource heterogeneity. Furthermore, we have proven the NP-completeness of *Multiple* for Replica Placement with QoS on homogeneous platforms, and we provided an optimal algorithm for the problem with QoS and bandwidth constraints on homogeneous platforms with the *Closest* policy.

On the practical side, we have designed several heuristics for the *Closest*, *Upwards* and *Multiple* policies, and we have compared their performance. To evaluate the absolute performance of our algorithms, we have compared the experimental results to the optimal solution of an integer linear program (ILP), and these results turned out quite satisfactory. Using our mono-criteria heuristics, the impact of the new policies is impressive: (i) the number of trees which admit a solution is much higher with the *Upwards* and *Multiple* policies than with the *Closest* policy; (ii) for those problems which have a solution with the *Closest* policy, the replica cost is much lower for the other two policies. In our experiments with QoS constraints, we have assessed the impact of QoS constraints on the different policies, and we have discussed which heuristic performed best depending upon problem instances, platform parameters and QoS tightness.

10.1.2 Pipeline Workflow Applications

In the second part, we have studied a difficult bi-criteria mapping problem onto *Fully Heterogeneous* platforms. We restricted ourselves to the class of applications which have a pipeline structure, and we have studied the complexity of the problem.

We have assessed the complexity of trading between period and latency, as well as between response time and reliability. These criteria are among the most important ones for a typical

user. Indeed, in the context of large scale distributed platforms such as clusters or grids, failure probability becomes a major concern [31, 33, 28], and the bi-criteria approaches tackled in this part enable to provide robust solutions while fulfilling user demands (minimizing latency under some reliability threshold, or the converse; minimizing period under a latency constraint, or the converse). We have shown that the more heterogeneity in the target platforms, the more difficult the problems.

In particular, the bi-criteria optimization problem for latency and failure probability is polynomial for *Fully Homogeneous*, NP-hard for *Fully Heterogeneous* and remains an open problem for *Communication Homogeneous*. Regarding period-latency optimization, the problem is polynomial as long as there is not any heterogeneity, hence it is NP-complete for *Communication Homogeneous* and *Fully Heterogeneous* platforms.

We provided several efficient polynomial heuristics for *Communication Homogeneous* platforms, either to minimize the period for a fixed latency, or to minimize the latency for a fixed period.

A typical application class, the digital image coding where images are processed in steady-state mode, has been considered as proof-of-concept. We have provided a detailed study of the bi-criteria mapping (minimizing period and latency) of the JPEG encoder application pipeline on a cluster of workstations. Experimental evaluations have then been carried out on generalized data sets and machine configurations, as well as for the JPEG encoder application.

The results of the general experiments show that the efficiency highly depends on platform parameters such as number of stages and number of available processors. Simple mono-criterion splitting heuristics perform very well when there is a limited number of processors, whereas bi-criteria heuristics perform much better when increasing the number of processors. Overall, the introduction of bi-criteria heuristics was not fully successful for small clusters but turned out to be mandatory to achieve good performance on larger platforms. Also, we have derived an ILP formulation of the bi-criteria optimization problem, which enabled us to assess the absolute performance of the heuristics. The results were quite satisfactory on those (not so many) instances for which we could compute the optimal solution in reasonable time. Finally, we observed that the 2-Split technique always returned the optimal solution for the JPEG encoder.

At the moment we are working on the study of the interplay between throughput, latency and reliability, a very challenging algorithmic problem. We plan to perform real-life experiments on heterogeneous platforms, using an already-implemented skeleton library, in order to compare the effective performance of the application for a given mapping (obtained with our heuristics) against the theoretical performance of this mapping.

10.1.3 Complex Streaming Applications

In the third part of this work, we have studied the operator mapping problem of in-network stream-processing applications onto a collection of heterogeneous processors. These stream-processing applications come as a set of operator trees, that have to continuously download basic objects at different sites of the network and at the same time have to process this data to produce some final result.

First we have formalized the operator-placement problem focusing on a "constructive" scenario in which one aims at minimizing the rental cost of a platform that satisfies an application throughput requirement. The complexity analysis showed that all problems are NP-complete, even for the simpler cases.

Second we have considered the problem for multiple concurrent applications under a non-constructive scenario, in which a fixed set of computation and communication resources is available and the goal is to minimize a cost function. Four different optimization problems were identified. All are NP-hard but can be formalized as integer linear programs.

In both cases we have proposed several polynomial heuristics and we evaluated these heuristics via extensive simulations. For single applications we assessed the absolute performance of our heuristics with respect to the optimal solution of the linear program for homogeneous platforms and small problem instances. The Subtree-bottom-up heuristic almost always produces optimal results and almost always outperforms the other heuristics. In the case of multiple concurrent applications our experiments showed the importance of node reuse across applications. Reusing nodes leads to an important number of additional solutions, and also the quality of the solutions improves considerably. We concluded that top-down traversals of the application trees is more efficient than bottom-up approaches, and in particular the combination of a top-down traversal with a breadth-first search (i.e., our heuristic TopDownBFS) achieved good results across the board.

10.2 Perspectives

In the following we detail possible extensions to the three investigated application classes.

10.2.1 Extensions for the Replica Placement Problem

In this work, we have considered a simplified instance of the replica problem and we outline two important generalizations, namely dealing with several objects, and changing the objective function.

With Several Objects

In this work, we have restricted the study of the problem to a single object, which means that all replicas are identical (of the same type). We can envision a system in which different types of objects need to be accessed. The clients are then having requests of different types, which can be served only by an appropriate replica. Thus, for an object of type k, client $i \in \mathcal{C}$ issues $r_i^{(k)}$ requests for this object. To serve a request of type k, a node must be provided with a replica of that type. Nodes can be provided with several replica types. A given client is likely to have different servers for different objects. The QoS may also be object-dependent ($\mathsf{q}_i^{(k)}$).

To refine further, new parameters can be introduce such as the size of object k and the computation time involved for this object. Nodes parameters become object-dependent too, in particular the storage cost and the time required to answer a request.

The server capacity constraint must then be a sum on all the object types, while the QoS must be satisfied for each object type. The link capacity also is a sum on the different object types, taking into account the size of each object.

There remains to modify the objective function: we simply aim at minimizing the cost of all replicas of different types that have been assigned to the nodes in the solution to get the extended *replica cost* for several objects.

Because the constraints add up linearly for different objects, it is not difficult to extend the linear programming formulation of Section 2.2 to deal with several objects. Also, the three access policies *Closest*, *Upwards* and *Multiple* could naturally be extended to handle several

objects. However, designing efficient heuristics for various object types, especially with different communication to computation ratios and different QoS constraints for each type, is a challenging algorithmic problem.

More Complex Objective Functions

Several important extensions of the problem consist in having a more complex objective function. In fact, either with on or with several objects, we have restricted so far to minimizing the cost of the replicas (and even their number in the homogeneous case). However, several other factors can be introduced in the objective function:

Communication cost – This cost is the *read* cost, i.e., the communication cost required to access the replicas to answer requests. It is thus a sum on all objects and all clients of the communication time required to access the replica. If we take this criteria into account in the objective function, we may prefer a solution in which replicas are close to the clients.

Update cost – The *write* cost is the extra cost due to an update of the replicas. An update must be performed when one of the clients is modifying (writing) some of the data. In this case, to ensure the consistency of the data, we need to propagate the modification to all other replicas of the modified object. Usually, this cost is directly related to the communication costs on the minimum spanning tree of the replica, since the replica which has been modified sends the information to all the other replicas.

Linear combination – A quite general objective function can be obtained by a linear combination of the three different costs, namely replica cost, read cost and write cost. As all three objectives are cost objectives, they can naturally be combined in a single objective function (in contrast to the antagonist criteria that was subject of this work). Informally, such an objective function would write

$$\alpha \sum_{\text{servers, objects}} \textit{replica cost} + \beta \sum_{\text{requests}} \textit{read cost} + \gamma \sum_{\text{updates}} \textit{write cost}$$

where the application-dependent parameters α, β and γ would be used to give priorities to the different costs.

Note that the extension to QoS constraints would imply to return to our threshold technique, where we fix one parameter and optimize for the other.

Again, designing efficient heuristics for such general objective functions, especially in the context of heterogeneous resources, is a challenging algorithmic problem.

10.2.2 Extensions for Workflow Applications

In future work, we could extend our heuristics to fully heterogeneous platforms, which appears to be challenging, even for a mono-criterion optimization problem. Indeed, because all link bandwidths are different, it seems hard to predict communication times as long as the mapping is not fully constructed. Thus, it is not easy to determine a strategy capable of simultaneously load balance computations while keeping communications under a prescribed threshold.

A natural extension of this work would be to consider other widely used skeletons. For example, when there is a bottleneck in the pipeline operation due to a stage which is both

computationally-demanding and not constrained by internal dependencies (such as the FDCT stage of the JPEG encoder), we can nest another skeleton in place of this stage. For instance a farm or deal skeleton would allow to split the workload of the initial stage among several processors. Using such deal skeletons may be either the programmer's decision (explicit nesting in the application code) or the result of the mapping procedure. Extending our mapping strategies to automatically identify opportunities for deal skeletons, and implement these, is a difficult but very interesting perspective.

10.2.3 Extensions for Concurrent Streaming Applications

In this work we have restricted the study to single applications in a constructive scenario and to multiple concurrent applications in non-constructive conditions. Of course, we can also extend our results and heuristics for multiple applications within a constructive scenario.

As future work, we could develop heuristics for the other optimization problems defined in Section 9.1.4. We could also envision a more general cost function $\mathsf{w}_{i,u}$ (time required to compute operator i onto processor u), in order to express even more heterogeneity. This would lead to the design of more sophisticated heuristics. Also, we believe it would be interesting to add a storage cost for objects downloaded onto processors, which could lead to new objective functions. Finally, we could address more complicated scenarios with many (conflicting) relevant criteria to consider simultaneously, some related to performance (throughput, response time), some related to safety (replicating some computations for more reliability), and some related to environmental costs (resource costs, energy consumption).

10.3 Final Remarks

It was both challenging and rewarding to devote three years to all these multi-criteria optimization problems. Here are some general concluding remarks from the lessons learnt during this thesis:

- As both applications and platforms get more complex, it is less likely that we find new polynomial problem instances. Combining many performance-related, safety-oriented or environment-aware objectives, we expect most problems to become NP-hard. On the theoretical side, we can hope for approximation algorithms, or at least for efficient heuristics whose absolute performance is assessed through an absolute ILP lower bound.

- Designing scheduling algorithms for platforms with heterogeneous communication links is an order of magnitude harder than for homogeneous links. This is because we no longer have an analytical estimation of the current cost of the objective function as long as the whole mapping is not finalized. We need new ideas and approaches here!

- Reliability issues arise in several contexts. Consider work-stealing approaches where different users suddenly reclaim loaned resources, according to probabilities that differ across user categories. This corresponds well to situations where failure-heterogeneous processors are expected to crash with different probability factors. It may be interesting to revisit both research areas and combine existing results.

Appendix A

Algorithms

Pseudo-code for Replica Placement heuristics

Algorithm 9: Procedure CTDA

```
procedure CTDA (root, replica)
Fifo fifo;
fifo.push(root);
while fifo ≠ ∅ do
    s = fifo.pop();
    if s ∉ replica then
        if W_s ≥ inreq_s & inreq_s > 0 then
            replica = replica ∪ {s};
            foreach a ∈ Ancestors(s) do inreq_a = inreq_a − inreq_s;
        else
            foreach i ∈ children(s) do
                if i ∈ 𝒩 then  fifo.push(i);
            end
        end
    end
end
```

Algorithm 10: Procedure CBU

```
procedure CBU (s ∈ N, replica)
if atBottom(s) || allChildrenTreated(s) then
    treated_s = true;
    if W_s ≥ inreq_s & inreq_s > 0 then
        /* node can treat all children's requests */
        replica = replica ∪ {s};
        foreach a ∈ Ancestors(s) do inreq_a = inreq_a − inreq_s;
    else
        /* node cannot treat all children's requests, go up in the tree */
        if Ancestors(s) ≠ ∅ then  call CBU (parent(s), replica);
    end
else
    foreach i ∈ children(s) do
        /* not yet at the bottom of the tree, go down */
        if i ∈ N & treated_i then  call CBU (i, replica);
    end
end
```

Algorithm 11: Procedure deleteRequests

```
procedure deleteRequests (s ∈ N, numToDelete)
clientList = sortDecreasing(clients(s));
foreach i ∈ clientList do
    if r_i ≤ numToDelete then
        numToDelete = numToDelete - r_i;
        foreach a ∈ Ancestors(i) do inreq_a = inreq_a − r_i;
        children(parent(i)) = children(parent(i)) \ {i};
        if numToDelete == 0 then  return;
    end
end
```

Algorithm 12: Procedure UTDFirstPass

```
procedure UTDFirstPass (s ∈ N, replica)
if inreq_s ≥ W_s & inreq_s > 0 then
    replica = replica ∪ {s};
    treated_s = true;
    deleteRequests(s, W_s);
end
foreach i ∈ children(s) do
    if i ∈ N then  UTDFirstPass (i, replica);
end
```

Algorithm 13: Procedure UTDSecondPass

```
procedure UTDSecondPass (s ∈ 𝒩, replica)
if s ∉ replica & inreq_s > 0 then
    replica = replica ∪ {s};
    deleteRequests(s, inreq_s);
else
    foreach i ∈ children(s) do
        if i ∈ 𝒩 & inreq_i > 0 then UTDSecondPass (i, replica);
    end
end
```

Algorithm 14: Procedure UBU

```
procedure UBU (s ∈ 𝒩, replica)
clientList = sortDecreasing(clients(s);
foreach i ∈ clientList do
    ValidAncests = {a ∈ Ancestors(i)|W_a ≥ r_i};
    if ValidAncests ≠ ∅ then
        a = Min_{W_j}{j ∈ ValidAncests};
        if a ∉ replica then replica = replica ∪ {a};
        W_a = W_a − r_i;
    end
    else return no solution;
end
```

Algorithm 15: Procedure deleteRequestsInMTD

```
procedure deleteRequestsInMTD (s ∈ 𝒩, numToDelete)
clientList = sortDecreasing(clients(s));
foreach i ∈ clientList do
    if r_i ≤ numToDelete then
        numToDelete = numToDelete - r_i;
        foreach a ∈ Ancestors(i) do inreq_a = inreq_a − r_i;
        children(parent(i)) = children(parent(i)) \ {i};
    else
        r_i = r_i - numToDelete;
        foreach a ∈ Ancestors(i) do inreq_a = inreq_a − r_i;
        return;
    end
end
```

Algorithm 16: Procedure MBUFirstPass

```
procedure MBUFirstPass (s ∈ 𝒩, replica)
if atBottom(s) || allChildrenTreated(s) then
    treated_s = true;
    if W_s ≤ inreq_s & inreq_s > 0 then
        /* node is exhausted by the requests of its clients */
        replica = replica ∪ {s};
        deleteRequestsInMBU(s, W_s);
    else
        /* node is not exhausted, go up the tree */
        if Ancestors(s) ≠ ∅ then  call MBU (parent(s), replica);
    end
else
    /* not yet at the bottom of the tree, go down */
    foreach i ∈ children(s) do
        if i ∈ 𝒩 & treated_i then  call MBU (i, replica);
    end
end
```

Algorithm 17: Procedure MBUSecondPass

```
procedure MBUSecondPass (s ∈ 𝒩, replica)
if s ∉ replica & inreq_s > 0 then
    replica = replica ∪ {s};
    deleteRequestsInMBU(s, inreq_s);
else
    foreach i ∈ children(s) do
        if i ∈ 𝒩 & inreq_i > 0 then  UTDSecondPass (i, replica);
    end
end
```

Appendix B

Bibliography

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January 2005.

[3] Jemal H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society Press, 2004.

[4] Luciano Volcan Agostini, Ivan Saraiva Silva, and Sergio Bampi. Parallel color space converters for JPEG image compression. *Microelectronics Reliability*, 44(4):697–703, 2004.

[5] Yanif Ahmad and Ugur Cetintemel. Network aware query processing for stream-based applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'04)*, pages 456–467, 2004.

[6] Susanne Albers and Günter Schmidt. Scheduling with unexpected machine breakdowns. *Discrete Applid Mathematics*, 110(2-3):85–99, 2001.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'04)*, pages 456–467, 2004.

[8] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[9] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In *HeteroPar'2004: International Conference on Heterogeneous Computing, jointly published with ISPDC'2004: International Symposium on Parallel and Distributed Computing*, pages 296–302. IEEE Computer Society Press, 2004.

[10] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. *Journal of Parallel and Distributed Computing*, 68(6):790–808, 2008.

[11] Anne Benoit, Yves Robert, and Eric Thierry. On the Complexity of Mapping Linear Chain Applications onto Heterogeneous Platforms. Research Report 2008-32, LIP laboratory, ENS Lyon, October 2008. Available at http://graal.ens-lyon.fr/~abenoit.

[12] Michael Beynon, Alan Sussman, Umit Catalyurek, Tahsin Kurc, and Joel Saltz. Performance optimization for data intensive grid applications. In *Proceedings of the Third Annual International Workshop on Active Middleware Services (AMS'01)*. IEEE Computer Society Press, 2001.

[13] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. *Future Generation Computer Systems*, 18(4):435–448, 2002.

[14] Prashanth B. Bhat, Cauligi S. Raghavendra, and Viktor K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[15] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Conference on Mobile Data Management*, 2001.

[16] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag Berlin Heidelberg, 2004.

[17] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *Maple Reference Manual*, 1988.

[18] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of the International Conference on Data Engineering (ICDE'02)*, 2002.

[19] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[20] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. GATES: a grid-based middleware for processing distributed data streams. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 192–201, 4-6 June 2004.

[21] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the CIDR Conference*, January 2003.

[22] Israel Cidon, Shay Kutten, and Ran Soffer. Optimal allocation of electronic content. *Computer Networks*, 40:205–218, 2002.

[23] Murray Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[24] Evan Cooke, Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs. Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[26] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: high-performance network monitoring with an SQL interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 623–633, 2002.

[27] DataCutter Project: Middleware for Filtering Large Archival Scientific Datasets in a Grid Environment. http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm.

[28] Angelo Duarte, Dolores Rexachs, and Emilio Luque. A distributed scheme for fault-tolerance in large clusters of workstations. In *NIC Series, Vol. 33*, pages 473–480. John von Neumann Institute for Computing, Julich, 2006.

[29] J. Falkemeier and G. Joubert. Parallel image compression with JPEG for multimedia applications. In Jack Dongarra et al., editor, *High Performance Computing: Technologies, Methods and Applications*, number 10 in Advances in Parallel Computing, pages 379–394. North Holland, 1995.

[30] Marco Ferretti and M. Boffadossi. A Parallel Pipelined Implementation of LOCO-I for JPEG-LS. In *17th International Conference on Pattern Recognition (ICPR'04)*, volume 1, pages 769–772, 2004.

[31] A. H. Frey and Geoffrey C. Fox. Problems and approaches for a teraflop processor. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 21–25. ACM Press, 1988.

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[33] Al Geist and Christian Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors. http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf, 2002.

[34] GLPK: GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[35] Bo Hong and Viktor K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society Press, 2004.

[36] Bo Hong and Viktor K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1420–1435, 2007.

[37] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'03)*, September 2003.

[38] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[39] Konstantinos Kalpakis, Koustuv Dasgupta, and Ouri Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):628–637, 2001.

[40] Konstantinos Kalpakis, Koustuv Dasgupta, and Ouri Wolfson. Steiner-Optimal Data Replication in Tree Networks with Storage Costs. In *Proceedings of the 2001 International Symposium on Database Engineering & Applications (IDEAS'01)*, pages 285–293. IEEE Computer Society Press, 2001.

[41] Magnus Karlsson and Christos Karamanolis. Choosing Replica Placement Heuristics for Wide-Area Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society Press.

[42] Magnus Karlsson, Christos Karamanolis, and Mallik Mahalingam. A framework for evaluating replica placement algorithms. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA, 2002.

[43] Jürgen Krämer and Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data streams. In *Proceedings of the 2005 SIGMOD International Confernece on Management of Data*, pages 70–82, 2005.

[44] Takeshi Kumaki, Masakatsu Ishizaki, Tetsushi Koide, Hans Jurgen Mattausch, Yasuto Kuroda, Hideyuki Noda, Katsumi Dosaka, Kazutami Arimoto, and Kazunori Saito. Acceleration of DCT Processing with Massive-Parallel Memory-Embedded SIMD Matrix Processor. *IEICE Transactions on Information and Systems - LETTER- Image Processing and Video Processing*, E90-D(8):1312–1315, 2007.

[45] Ling Liu, Calton Pu, and Wei Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[46] Pangfeng Liu, Yi-Fang Lin, and Jan-Jan Wu. Optimal placement of replicas in data grid environments with locality assurance. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society Press, 2006.

[47] Dionysios Logothetis and Kenneth Yocum. Wide-Scale Data Stream Management. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[48] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 491–502, 2003.

[49] Peter Meerwald, Roland Norcen, and Andreas Uhl. Parallel JPEG2000 Image Coding on Multiprocessors. In *International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society Press, 2002.

[50] Peter Monnes and Borko Furht. Parallel JPEG Algorithms for Still Image Processing. In *Proceedings of the 1994 IEEE SoutheastCon. Creative Technology Transfer - A Global Affair.*, pages 375 – 379, apr 1994.

[51] Suman Nath, Amol Deshpande, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. IrisNet: An Architecture for Internet-scale Sensing Services.

[52] Vinayaka Pandit and Huibo Ji. Efficient In-Network Evaluation of Multiple Queries. In *International Conference on High Performance Computing (HiPC'06)*, 2006.

[53] Markos Papadonikolakis, Vasilleios Pantazis, and Athanasios P. Kakarountas. Efficient high-performance ASIC implementation of JPEG-LS encoder. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE2007)*, volume IEEE Communications Society Press, 2007.

[54] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–60, 2006.

[55] Ali Pinar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[56] Beth Plale and Karsten Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, 2003.

[57] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[58] Georgios Rodolakis, Stavroula Siachalou, and Leonidas Georgiadis. Replicated server placement with QoS constraints. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1151–1162, 2006.

[59] Taher Saif and Manish Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182. Springer Verlag, 2004.

[60] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[61] Seng Lin Shee, Andrea Erdos, and Sri Parameswaran. Architectural Exploration of Heterogeneous Multiprocessor Systems for JPEG. *International Journal of Parallel Programming*, 35, 2007.

[62] Ke Shen, Gregory W. Cook, Leah H. Jamieson, and Edward J. Delp. An overview of parallel processing approaches to image and video compression. In M. Rabbani, editor, *Image and Video Compression*, volume Proc. SPIE 2186, pages 197–208, 1994.

[63] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, 1995.

[64] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *2002 ACM/IEEE Supercomputing Conference*. ACM Press, 2002.

[65] Utkarsh Srivastava and Kamesh Munagala Jennifer Widom. Operator Placement for In-Network Stream Query Processing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'05)*, pages 250–258, 2005.

[66] Jaspal Subhlok and Gary Vondran. Optimal mapping of sequences of data parallel tasks. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 134–143. ACM Press, 1995.

[67] Jaspal Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures SPAA'96*, pages 62–71. ACM Press, 1996.

[68] Xueyan Tang and Jianliang Xu. QoS-Aware Replica Placement for Content Distribution. *IEEE Transactions on Parallel and Distributed Systems*, 16(10):921–932, 2005.

[69] Kenjiro Taura and Andrew A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW'2000, the 9th Heterogeneous Computing Workshop*, pages 102–115. IEEE Computer Society Press, 2000.

[70] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[71] Robbert van Rennesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. Scalable Management and Data Mining Using Astrolabe. In *Proceedings from the First International Workshop on Peer-to-Peer Systems*, pages 280–294, 2002.

[72] Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tashin Kurc, P. Saddayappan, and Joel Saltz. An approach for optimizing latency under throughput constraints for application workflows on clusters. Research Report OSU-CISRC-1/07-TR03, Ohio State University, Columbus, OH, January 2007. Available at ftp://ftp.cse.ohio-state.edu/pub/tech-report/2007. Short version appears in EuroPar'2008.

[73] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[74] Hsiangkai Wang, Pangfeng Liu, , and Jan-Jan Wu. A QoS-aware Heuristic Algorithm for Replica Placement. In *Proceedings of the 7th International Conference on Grid Computing (GRID2006)*, pages 96–103. IEEE Computer Society Press, 2006.

[75] Chen Wen-Hsiung, C. Smith, and Stanley Fralick. A Fast Computational Algorithm for the Discrete Cosine Transform. *IEEE Transactions on Communications*, 25(9):1004–1009, 1977.

[76] Ouri Wolfson and Amir Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Database Systems*, 16(1):181–205, 1991.

[77] Ruibin Xu, Rami Melhem, and Daniel Mosse. Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors. In *Proceedings of the 28th IEEE Real-Time System Symposium (RTSS'07), Tucson, Arizona*, December 2007.

Appendix C

Publications

The publications are listed in reverse chronological order.

Articles in international refereed journals

[A1] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria Scheduling of Pipeline Workflows (and Application to the JPEG Encoder). *International Journal of High Performance Computing Applications*, 23:171 – 187, may 2009.

[A2] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Replica Placement and Access Policies in Tree Networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1614 – 1627, dec 2008.

[A3] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling algorithms for data redistribution and load-balancing on master-slave platforms. *Parallel Processing Letters*, 17(1), 2007.

Articles in international refereed conferences

[B1] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves Robert. Resource allocation strategies for constructive in-network stream processing. In *APDCM'09, the 11th Workshop on Advances in Parallel and Distributed Computational Models*. IEEE Computer Society Press, 2009.

[B2] Anne Benoit, Harald Kosch, Veronika Rehn-Sonigo, and Yves Robert. Bi-criteria Pipeline Mappings for Parallel Image Processing. In *ICCS'08, the 2008 International Conference on Computational Science*. Springer Verlag, 2008.

[B3] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Optimizing Latency and Reliability of Pipeline Workflow Applications. In *HCW'08, the 17th Heterogeneity in Computing Workshop*. IEEE Computer Society Press, 2008.

[B4] Anne Benoit, Veronika Rehn-Sonigo, and Yves Robert. Multi-criteria Scheduling of Pipeline Workflows. In *HeteroPar'07, Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (in conjunction with Cluster 2007)*. IEEE Computer Society Press, 2007.

[B5] Veronika Rehn-Sonigo. Optimal Closest Policy with QoS and Bandwidth Constraints for Placing Replicas in Tree Networks. In *CoreGRID'2007, Core GRID Symposium 2007*. Springer Verlag, 2007.

[B6] Anne Benoit, Veronika Rehn, and Yves Robert. Impact of QoS on Replica Placement in Tree Networks. In *ICCS'2007, the 2007 International Conference on Computational Science*. Springer Verlag, 2007.

[B7] Anne Benoit, Veronika Rehn, and Yves Robert. Strategies for Replica Placement in Tree Networks. In *HCW'2007, the 16th Heterogeneity in Computing Workshop*. IEEE Computer Society Press, 2007.

[B8] Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling and data redistribution strategies on star platforms. In *PDP'2007, 15th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE Computer Society Press, 2007.

[B9] Olivier Beaumont, Loris Marchal, Veronika Rehn, and Yves Robert. Fifo scheduling of divisible loads with return messages under the one-port model. In *HCW'2006, the 15th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2006.

Appendix D

Notations

Replica Placement in Tree Networks

Chapters 1 - 3

$\mathcal{T} = \mathcal{N} \cup \mathcal{L}$	distribution tree $\mathcal{T}$
$\mathcal{N}$	set of internal nodes (possible servers)
$\mathcal{C}$	set of leaf nodes (clients)
$\mathcal{L}$	set of tree edges (links)
r_i	requests of client i, $i \in \mathcal{C}$
q_i	QoS requirement for requests of client i
W_j	processing capacity of node j, $j \in \mathcal{N}$
sc_j	price to pay to place a replica at node j (storage cost)
$\mathsf{Servers}(i) \subseteq \mathcal{N}$	set of nodes provided with a replica and treating the requests of client i
comm_l	communication time on link l
BW_l	bandwidth of link l
r	root of the tree
$\mathsf{children}(j)$	set of children of node j, $j \in \mathcal{N}$
$\mathsf{parent}(k)$	parent of node k, $k \in \mathcal{N} \cup \mathcal{C}$
$\mathsf{succ}(l)$	if link l connects node k to $k' = \mathsf{parent}(k)$, $\mathsf{succ}(l)$ denotes the link that connects k' to $\mathsf{parent}(k')$
$\mathsf{Ancestors}(k)$	ancestors of node k, i.e., the nodes in the unique path that leads from k up to the root r (k excluded)
$\mathsf{path}[k \rightarrow k']$	path from k to k'
R	set of replicas

Problems

Replica Placement	general problem
Replica Counting	homogeneous nodes
Replica Cost	heterogeneous nodes
Replica Counting with QoS	homogeneous nodes, QoS constraints
Replica Cost with QoS	heterogeneous nodes, QoS constraints
Replica Cost with QoS and Bandwidth	homogeneous nodes, QoS and bandwidth constraints

Section 3.1.2

T^*	tree
r^+	root
$t(v)$	subtree rooted by node v
$t'(v) = t(v) - v$	forest of trees rooted at v's children
$a(v, i)$	i'th ancestor of node v, traversing the tree up to the root
$m(T^*)$	minimum cardinality set of replicas that has to be placed in tree T
W	processing capacity of a node
$m(t(v))$	minimum number of replicas that has to be placed in $t'(v)$
$C(v, i)$	minimum number of requests on node $a(v, i)$ contributed by $t(v)$ by placing $m(t(v))$ replicas in $t'(v)$ and none on $a(v, j)$ for $0 \leq j < i$
$e(v, i)$	children of node v that have to be equipped with a replica such that the remaining requests on node $a(v, i)$ are within W, there are exactly $m(t(v))$ replicas in $t'(v)$ and none on $a(v, j)$ for $0 \leq j < i$ and the contribution $t(v)$ on $a(v, i)$ is minimized

Pipeline Workflow Applications

Chapters 4 - 6

n	number of stages
$\mathcal{S}_k$	stage k
δ_k	data input for stage k
w_k	number of computations of stage k
p	number of processors
P_u	processor u
fp_u	failure probability of P_u
s_u	speed of processor P_u
$\mathsf{link}_{u,v}$	bidirectional link between P_u and P_v
$\mathsf{b}_{u,v}$	bandwidth of link $\mathsf{link}_{u,v}$
$\mathcal{L}$	latency
$\mathcal{P}$	period
$\mathcal{FP}$	failure probability

Concurrent Streaming Applications

Chapter 8

$\mathcal{N}$	set of operators
n_i	operator i
w_i	computation amount for n_i
δ_i	output size of n_i
$\mathcal{OB}$	set of basic objects
ob_k	object k
δ_k	size of ob_k
f_k	frequency of ob_k
$rate_k$	bandwidth consumption for the download of ob_k
$Leaf(i)$	index set of the basic objects needed for the computation of n_i, if any
$Child(i)$	the index set of n_i's children in $\mathcal{N}$, if any
$Parent(i)$	the index of n_i's parent in $\mathcal{N}$, if any
$\mathcal{P}$	set of processors
$\mathcal{S}$	set of servers
R	set of resources $\mathcal{P} \cup \mathcal{S}$
S_l	server l
P_u	processor u
Bs_l	network card bandwidth of S_l
Bp_u	network card bandwidth of P_u
bp	bandwidth between two processors
bs_l	bandwidth between S_l and a processor
s_u	compute speed of P_u
$a(i) = u$	allocation function, denotes that n_i is assigned to P_u
$\bar{a}(u)$	index set of operators mapped on P_u
$DL(u)$	set of (k, l) couples where P_u downloads ob_k from S_l
ρ	application throughput

Chapter 9

$\mathcal{K}$ number of applications

$\mathcal{OP}$ set of operators
op_p operator p
$operators(p)$ the index set of operators in $\mathcal{OP}$ whose intermediate results are needed for the computation of op_p, if any

$\mathcal{OB}$ set of basic objects
ob_j object j
$objects(p)$ index set of the basic objects in $\mathcal{OB}$ that are needed for the computation of op_p, if any

$n_i^{(k)}$ i-th internal node in the tree of application k
$op(n_i^{(k)})$ associated operator to $n_i^{(k)}$
$ob(n_i^{(k)})$ set of basic objects required by $op(n_i^{(k)})$

δ_j size of ob_j
$f_j^{(k)}$ download frequency of ob_j in application k
$rate_j^{(k)}$ bandwidth consumption for the download of ob_j for application k

w_p computation amount for op_p
δ_p output size of op_p

$\mathcal{P}$ set of processors
P_u processor u
Bp_u network card bandwidth of P_u
$bp_{u,v}$ bandwidth of the network link between P_u and P_v
s_u compute speed of p_u

$a(k,i) = u$ allocation function, denotes that $n_i^{(k)}$ is mapped on P_u
$\bar{a}(u)$ index set of operators mapped on P_u

$\rho^{(k)}$ throughput of application k

$Ch(u) = \{(p,v,k)\}$ set of (operator, processor, application) tuples such that processor P_u needs to receive an intermediate result computed by operator op_p, which is mapped to processor P_v, at rate $\rho^{(k)}$;
operators op_p are children of $a_{op}(u)$ in the operator tree.
$Par(u) = \{(p,v,k)\}$ set of (operator, processor, application) tuples such that P_u needs to send to P_v an intermediate result computed by operator op_p at rate $\rho^{(k)}$;
$p \in a_{op}(u)$ and the sending is done to the parents of op_p in the operator tree.
$DL(u) = \{(j,v,k)\}$ set of (object, processor, application) tuples where P_u downloads object ob_j from processor P_v at rate $\rho^{(k)}$.

Zusammenfassung:

Diese Arbeit befasst sich mit dem Scheduling von Workflow-Anwendungen in heterogenen Plattformen. In diesem Zusammenhang betrachten wir drei verschiedene Arten von Anwendungen:
Platzierung von Replikaten in Baumnetzwerken– Das erste Schedulingproblem interessiert sich für die Platzierung von Replikaten in Baumnetzwerken. Ein Beispiel hierfür ist die Platzierung von Replikaten in verteilten Datenbanksystemen, deren Verbindungsstruktur baumartig organisiert ist. Die Platzierung soll unter mehreren Constraints (Serverkapazitäten, sowie Dienstgüte und Bandbreitenbeschränkungen) durchgeführt werden. Die Client-Anfragen sind im Voraus bekannt, während Anzahl und Platzierung der Server erst zu ermitteln sind. Die in der Literatur gängige Strategie erzwingt, dass alle Anfragen eines Clients vom nächstgelegenen Server im Baum behandelt werden. Es werden zwei neue Verfahrensweisen vorgestellt und untersucht. Ein Hauptbeitrag besteht in der Bewertung der Auswirkung der beiden neuen Strategien auf die globalen Replikationskosten. Ein weiteres wichtiges Ziel ist die Behandlung von Heterogenität aus theoretischer und praktischer Sicht. Es werden verschiedene Komplexitätsergebnisse erarbeitet und mehrere effiziente Polynomialzeit-Heuristiken für NP-vollständige Instanzen des Problems vorgestellt.
Lineare Workflow-Anwendungen– Als nächstes werden Workflow-Anwendungen behandelt, welche als lineare Graphen dargestellt werden können. Ein Beispiel dieses Applikationstyps ist die digitale Bildverarbeitung, in der Bilder einer Pipeline verarbeitet werden. Dabei sollen verschiedene gegensätzliche Kriterien optimiert werden, wie zum Beispiel Durchsatz und Latenzzeit, beziehungsweise eine Kombination der beiden, oder auch Latenzzeit und Ausfallsicherheit der Anwendung (d.h. die Wahrscheinlichkeit, dass die Verarbeitung erfolgreich ist). Während für vollhomogene Plattformen polynomiale Algorithmen gefunden werden können, wird das Problem NP-hart, sobald heterogene Plattformen angestrebt werden. Ein ganzzahliges lineares Programm für letzteres Problem wird vorgestellt. Des weiteren werden verschiedene effiziente polynomiale bi-kritäre Heuristiken präsentiert, deren relative Effizienz durch umfangreiche Simulationen eruiert werden. In einer Fallstudie werden Simulationen und MPI-basierte Auswertungen für die JPEG-Encoder-Pipeline auf einem Rechen-Cluster erstellt.
Komplexe Streaming-Anwendungen– Als letztes wird die Ausführung von Anwendungen, die als Operator-Bäume strukturiert sind, untersucht. Konkret bedeutet dies, dass ein oder mehrere Operator-Bäume in stationärem Zustand auf mannigfaltige Datenobjekte angewendet werden, welche fortlaufend an verschiedenen Stellen im Netzwerk aktualisiert werden. Ein erstes Ziel ist es, dem Benutzer eine Gruppe von Rechnern vorzuschlagen, die gekauft oder gemietet werden sollen, so dass die Anwendung einen minimalen stationären Durchsatz erzielt und gleichzeitig Plattformkosten minimiert werden können. Später wird das Modell auf mehrere Anwendungen erweitert: verschiedene nebenläufige Anwendungen werden zeitgleich in einem Netzwerk ausgeführt und es gilt sicherzustellen, dass alle Anwendungen ihren Durchsatz erreichen können. Ein weiterer Beitrag dieser Arbeit ist die Erstellung einer Komplexitätsanalyse für unterschiedliche Instanzen des Grundproblems, sowie die Formulierung verschiedener Instanzen als lineare Programme. Als dritten Beitrag werden für beide Anwendungsmodelle verschiedene Polynomialzeit-Heuristiken präsentiert. Ein Hauptziel der Heuristiken für nebenläufige Anwendungen ist die Wiederverwertung von Zwischenergebnissen, welche von mehreren Anwedungen geteilt werden.

Schlüsselworte:
Platzierung von Replikaten, Pipeline, Flussverarbeitung im Netzwerk, Platzierung von Operatoren, Baumnetzwerke, multi-kritäre Optimierung, Komplexitätsanalyse, Heuristiken, lineare Programme, heterogene Plattformen.

Résumé :

Les travaux présentés dans cette thèse portent sur le placement et l'ordonnancement d'applications de flux de données sur des plates-formes hétérogènes. Dans ce contexte, nous nous concentrons sur trois types différents d'applications :

Placement de répliques dans les réseaux hiérarchiques– Dans ce type d'application, plusieurs clients émettent des requêtes à quelques serveurs et la question est : où doit-on placer des répliques dans le réseau afin que toutes les requêtes puissent être traitées. Nous discutons et comparons plusieurs politiques de placement de répliques dans des réseaux hiérarchiques en respectant des contraintes de capacité de serveur, de qualité de service et de bande-passante. Les requêtes des clients sont connues a priori, tandis que le nombre et la position des serveurs sont à déterminer. L'approche traditionnelle dans la littérature est de forcer toutes les requêtes d'un client à être traitées par le serveur le plus proche dans le réseau hiérarchique. Nous introduisons et étudions deux nouvelles politiques. Une principale contribution de ce travail est l'évaluation de l'impact de ces nouvelles politiques sur le coût total de replication. Un autre but important est d'évaluer l'impact de l'hétérogénéité des serveurs, d'une perspective à la fois théorique et pratique. Nous établissons plusieurs nouveaux résultats de complexité, et nous présentons plusieurs heuristiques efficaces en temps polynomial.

Applications de flux de données– Nous considérons des applications de flux de données qui peuvent être exprimées comme des graphes linéaires. Un exemple pour ce type d'application est le traitement numérique d'images, où les images sont traitées en régime permanent. Plusieurs critères antagonistes doivent être optimisés, tels que le débit et la latence (ou une combinaison) ainsi que la latence et la fiabilité (i.e. la probabilité que le calcul soit réussi) de l'application. Bien qu'il soit possible de trouver des algorithmes polynomiaux simples pour les plates-formes entièrement homogènes, le problème devient NP-difficile lorsqu'on s'attaque à des plates-formes hétérogènes. Nous présentons une formulation en programme linéaire pour ce dernier problème. De plus nous introduisons plusieurs heuristiques bi-critères efficaces en temps polynomial, dont la performance relative est évaluée par des simulations extensives. Dans une étude de cas, nous présentons des simulations et des résultats expérimentaux (programmés en MPI) pour le graphe d'application de l'encodeur JPEG sur une grappe de calcul.

Applications complexes de streaming– Considérons l'exécution d'applications organisées en arbres d'opérateurs, i.e. l'application en régime permanent d'un ou plusieurs arbres d'opérateurs à données multiples qui doivent être mis à jour continuellement à différents endroits du réseau. Un premier but est de fournir à l'utilisateur un ensemble de processeurs qui doit être acheté ou loué pour garantir que le débit minimum de l'application en régime permanent soit atteint. Puis nous étendons notre modèle aux applications multiples : plusieurs applications concurrentes sont exécutées en même temps dans un réseau, et on doit assurer que toutes les applications puissent atteindre leur débit requis. Une autre contribution de ce travail est d'apporter des résultats de complexité pour des instances variées du problème. La troisième contribution est l'élaboration de plusieurs heuristiques polynomiales pour les deux modèles d'application. Un objectif premier des heuristiques pour applications concurrentes est la réutilisation des résultats intermédiaires qui sont partagés parmi différentes applications.

Mots-clés :

Placement de répliques, graphe d'application linéaire, traitement de flux de données, placement d'opérateurs, réseaux hiérarchiques, optimisation multi-critère, résultats de complexité, heuristiques, programme linéaire, plates-formes hétérogènes.

Abstract:
The results summarized in this document deal with the mapping and scheduling of workflow applications on heterogeneous platforms. In this context, we focus on three different types of streaming applications:
Replica placement in tree networks– In this kind of application, clients are issuing requests to some servers and the question is where to place replicas in the network such that all requests can be processed. We discuss and compare several policies to place replicas in tree networks, subject to server capacity, Quality of Service (QoS) and bandwidth constraints. The client requests are known beforehand, while the number and location of the servers have to be determined. The standard approach in the literature is to enforce that all requests of a client be served by the closest server in the tree. We introduce and study two new policies. One major contribution of this work is to assess the impact of these new policies on the total replication cost. Another important goal is to assess the impact of server heterogeneity, both from a theoretical and a practical perspective. We establish several new complexity results, and provide several efficient polynomial heuristics for NP-complete instances of the problem.
Pipeline workflow applications– We consider workflow applications that can be expressed as linear pipeline graphs. An example for this application type is digital image processing, where images are treated in steady-state mode. Several antagonist criteria should be optimized, such as throughput and latency (or a combination) as well as latency and reliability (i.e., the probability that the computation will be successful) of the application. While simple polynomial algorithms can be found for fully homogeneous platforms, the problem becomes NP-hard when tackling heterogeneous platforms. We present an integer linear programming formulation for this latter problem. Furthermore, we provide several efficient polynomial bi-criteria heuristics, whose relative performances are evaluated through extensive simulation. As a case-study, we provide simulations and MPI experimental results for the JPEG encoder application pipeline on a cluster of workstations.
Complex streaming applications– We consider the execution of applications structured as trees of operators, i.e., the application of one or several trees of operators in steady-state to multiple data objects that are continuously updated at various locations in a network. A first goal is to provide the user with a set of processors that should be bought or rented in order to ensure that the application achieves a minimum steady-state throughput, and with the objective of minimizing platform cost. We then extend our model to multiple applications: several concurrent applications are executed at the same time in a network, and one has to ensure that all applications can reach their application throughput. Another contribution of this work is to provide complexity results for different instances of the basic problem, as well as integer linear program formulations of various problem instances. The third contribution is the design of several polynomial-time heuristics, for both application models. One of the primary objectives of the heuristics for concurrent applications is to reuse intermediate results shared by multiple applications.
Keywords:
Replica placement, pipeline, in-network stream processing, operator mapping, tree networks, multi-criteria optimization, complexity results, heuristics, linear program, heterogeneous platforms.