In this dissertation, we present several techniques to achieve the high-performance processing in networked and grid environments. Many applications need a high-performance processing system to execute efficiently. High-performance processing mainly stems from parallelism. The parallel nature of grid computing is a very attractive solution to exploit the mentioned parallelism by executing either different parts of an application or several applications in parallel. In a grid system, the most important resources are computing and communication resources. The computing resources are the processors in the nodes on the grid. Communication within the grid is important for distributing tasks and their required data to the nodes within the grid.

Delft University of Technology

ISBN: 978-90-72298-06-5

High-performance Processing in Networked and Grid Environments Mahmood Ahmadi

High-performance Processing in Networked and Grid Environments

Mahmood Ahmadi

PROPOSITIONS

MAHMOOD AHMADI

(1) A high performance processing system must be boundless like an ocean.

(2) A problem can be completely solved if it can be modeled, simulated, and implemented [*This thesis, Chapter 3, 4, 5].*

(3) Doing research in PhD is like climbing a mountain. You only think to rise.

(4) Existence of a false positive in Bloom filters shows that achieving the ideal is not possible, instead one should aim at the near-optimal point *[This thesis, Chpater 4].*

(5) An excellent research paper is like a famous song. The latter needs a tune, a singer, and a poem like a paper needing an idea, an implementation, and a presentation.

(6) High-performance computing is more of an art than a science.

(7) In the future, all processors will be reconfigurable *[This thesis, Chapter 3].*

(8) Bloom filters became popular 30 years after they were proposed by Burton Bloom in 1970. Many good ideas are still to be (re-)discovered *[This thesis, Chapter 4].*

(9) In order for all PhD students in the Netherlands to finish their study in 4 years, a radical change in the university system is required.

(10) A PhD thesis is like a mirror: your reflection can be seen in it.

These propositions are considered defendable and opposable and as such have been approved by Prof. dr. C.I.M. Beenakker.

(1) Een hoog-presterend verwerkingssysteem moet grenzeloos zijn als een oceaan.

(2) Een vraagstuk kan geheel worden opgelost als het gemodeleerd, gesimuleerd, en gemplementeerd kan worden *[Dit proefschrift, Hoofdstuk 3, 4, 5]*.

(3) Onderzoek tijdens je promotie is als het beklimmen van een berg. Je denkt alleen aan stijgen.

(4) Het bestaan van een fout-positief in Bloom-filters laat zien dat het niet mogelijk is het ideaal te bereiken, in plaats daarvan zou men moeten streven naar het optimum *[Dit proefschrift, Hoofdstuk 4]*.

(5) Een uitmuntend onderzoeksartikel is als een beroemd liedje. Het laatste heeft melodie, zanger en gedicht nodig en het eerste bevat een idee, implementatie en presentatie.

(6) 'High-performance computing' is meer een kunst dan een wetenschap.

(7) In de toekomst zullen alle processoren herconfigureerbar worden *[Dit proefschrift, Hoofdstuk 3]*.

(8) Bloom-filters werden 30 jaar nadat ze werden voorgesteld door Burton Bloom in 1970, populair. Veel goede ideeën moeten opnieuw ontdekt worden *[Dit proefschrift, Hoofdstuk 4]*.

(9) Om te zorgen dat alle promovendi binnen vier jaar hun promotie behalen, is een radicale verandering in het universiteits systeem nodig.

(10) Een dissertatie is als een spiegel: je spiegelbeeld kan er in worden gezien.

Deze stellingen worden verdedigbaar en opponeerbaar geacht en zijn zodanig goedgekeurd door Prof. dr. C.I.M. Beenakker.

High-performance Processing in Networked and Grid Environments

High-performance Processing In Networked and Grid Environments

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 18 may 2010 om 12:30 uur

door

Mahmood AHMADI

Master of science in Computer Engineering-Computer Architecture
Amirkabir University of Technology (Tehran Polytechnique), Iran
geboren te Kohdasht, Lorestan, Iran

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. C. I. M. Beenakker

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. C. I. M. Beenakker, promotor	Technische Universiteit Delft
Prof. dr. ir. I. Niemegeers	Technische Universiteit Delft
Prof. dr. S. Yalamanchili	Georgia Institute of Technology
Prof. dr. K. G. W. Goossens	Eindhoven University of Technology
Prof. dr. F. Safaei	Wolongong University, Australia
Prof. dr. J. Takala	Tampere University of Technology, Finland
Dr. Ir. S. Wong	Technische Universiteit Delft
Prof. dr. P.M. Sarro, reservelid	Technische Universiteit Delft

ISBN 978-90-72298-06-5

Cover page: “A grid network of high performance computing systems”

Keywords: Reconfigurable architecture, Bloom filter, grid computing

This dissertation is dedicated to Vahideh
and my family,
for all their understanding and support over the years.

High-performance Processing in Networked and Grid Environments

Mahmood Ahmadi

Abstract

In this dissertation, we present several techniques to achieve the high-performance processing in networked and grid environments. Many applications need a high-performance processing system to execute efficiently. High-performance processing mainly stems from parallelism. The parallel nature of grid computing is a very attractive solution to exploit the mentioned parallelism by executing either different parts of an application or several applications in parallel. In a grid system, the most important resources are **computing** and **communication** resources. The computing resources are the processors in the nodes on the grid. Communication within the grid is important for distributing tasks and their required data to the nodes within the grid.

We propose an innovative high-performance platform to utilize reconfigurable processors in grid environments. Furthermore, we focus on the communication infrastructures and network processing (processing required for packets) platforms to utilize them through the grid environments. The collaboration of reconfigurable processors in a grid environment is presented and several compute-intensive multimedia kernels are simulated. Subsequently, we introduce three approaches to accelerate network processing tasks using Bloom filters in the networked and grid environments. The first and second techniques present a cache architecture for a counting Bloom filter (CCBF) and a memory optimization approach for Bloom filters using an additional hashing function (BFAH). The third technique proposes a power efficient pipelined Bloom filter.

We present the results of our proposed approaches in collaboration of reconfigurable processors in grid computing (CRGC) and Bloom filters in network processing applications, e.g., packet classification. The experimental results show that the CRGC approach improves performance up to 7.2x and 2.5x compared to a GPP and the collaboration of GPPs, respectively. The results of the CCBF and BFAH for packet classification show that the proposed techniques decrease the number of memory accesses when compared to a standard Bloom filter.

Acknowledgments

First of all, I am most grateful to Stamatis Vassiliadis for believing in me and giving me the chance to do my PhD in his research group. It was my privilege to work under his supervision, although for such a short period of time. I will miss him and remember him always.

Equally, I would like, to thank Stephan Wong for his guidance and assistance. His help in reviewing my research work and improve my writing skills was extremely vital. My PhD work would not have been the same without his inputs. I acknowledge Prof. Ir. C. I. M. Beenakker, my promotor at DIMES. I wish to thank Prof. Dr. Kees Goossens and Prof. Jarmo Takala, the members of committee, for their useful comments on this thesis.

I would like to thank everybody from the Computer Engineering Laboratory where I had the opportunity to work in a truly international environment, with people coming from all parts of the world, with different work experiences.

I would like to thank Faisal M Nadeem for his time spent to read my thesis and Roel Meeuws and Core Meendrek for their time to translate the abstract and propositions to Dutch. I am thankful to Lidwina, Bert Meijs and Erick for the help and time they gave to me.

I would also like to acknowledge my Iranian friends Asadollah Shahbahrami, Mojtaba Sabeghi, and Arash Ostadzadeh at Computer Engineering Laboratory.

I would like to express my deepest gratitude to my parents, my brother and my sisters, for their love, trust, advices, and support during the entire life of mine.

Finally, special thanks go to Vahideh for her understanding, patience and support when I had to work during many weekends over the years.

M. Ahmadi Delft, The Netherlands, 2010

Contents

List of Figures

List of Tables

List of Acronyms

ACC Adaptable Computing Cluster

ANPQ Abstract Network Processor Queuing model

ASIC Application-Specific Integrated Circuits

ASIP Application-Specific Instruction-set Processor

ASAN Active System Area Network

AQRGS The Architecture for QoS-enable Router and Grid-oriented Supercomputer

BF Bloom Filters

BFAH Bloom Filters using an Additional Hashing Function

BOINC Berkeley Open Infrastructure for Network Computing

CRGC Collaboration of Reconfigurable processing elements in Grid Computing

CAM Content Addressable Memory

CRGridSim Collaborative Reconfigurable Grid Simulator

CCBF Cached Counting Bloom Filter

DRMC Distributed Reconfigurable Metacomputing

DPN Distributed Processing Network

DWT Discrete Wavelet Transform

FPGA Field Programmable Gate Arrays

FMO Fragment Molecular Orbital

GPP General-Purpose Processors

GRB Gamma-Rate Burst

ILP Instruction-Level Parallelism

MBHT Multi-predicate Bloom-filtered Hash Table

MVA Mean Value Analysis

MIPS Million Instructions Per Second

NP Network Processors

PLP Packet-Level Parallelism

PBF Pipelined Bloom Filter

P2P Peer to Peer

PE Processing Engine

RE Reconfigurable Element

RIP Routing Information Protocol

SIA Synchronous Iterative Algorithm

SOC System On Chip

SIMD Single-Instruction Multiple-Data

SPU Synergistic Processor Unit

SPE Synergistic Processing Element

SPEC Standard Performance Evaluation Corporation

TTM Time To Market

TTL Time To Live

TLP Thread-Level Parallelism

VLIW Very Long Instruction Word

VPN Virtual Private Network

Chapter 1

Introduction

The *performance requirement of many scientific applications is rising each year. Among these that require the highest are the following: weather forecasting, modeling in material sciences, applied fluid dynamics, ecosystem simulations, biomedical imaging and biomechanics, molecular design and process optimization, nuclear power and weapons simulations, design of pharmaceutical drugs, human genome, astronomy, numbers theory, computational ocean sciences, speech and vision. Much research effort has been put in building large-scale systems to efficiently compute these kind of applications. In such systems, it is important to focus on (1) the computing resources and (2) the communication resources. Therefore, designing and implementing more powerful approaches to combine the computing techniques and communication together is an efficient way to achieve higher performance.*

We motivate in Section 1.1, the motivation about some computationally intensive applications, their requirements, current high-performance processing systems, and related networking applications. In Section 1.2, the challenges and goals addressed in this dissertation are described and in Section 1.3, our methodology is introduced. Finally, Section 1.4 describes the outline of the dissertation.

1.1 Motivation

In this section, we present some instances of high performance applications, the existing and future supercomputers and the requirement in the high performance processing.

1.1.1 High-performance Applications

The ongoing demand for faster and high-performance computing systems is expected to continue in the coming years due to the increasing complexity and size of computationally intensive applications. Examples of these applications are: weather forecasting, modeling in material sciences, applied fluid dynamics, ecosystem simulations, biomedical imaging and biomechanics, molecular design and process optimization, nuclear power and weapons simulations, design of pharmaceutical drugs, human genome, astronomy, numbers theory, computational ocean sciences, speech and vision. The common characteristics of these applications is their inherent level of parallelism. It is this inherent parallelism that can be exploited to design ever-increasingly more powerful computing systems to execute these applications. Some instances of these applications with the number of operations required per simulation and required high-performance computer systems are presented in the following.

In global climate modeling[1] the calculation a series of factors in order to make them sufficiently accurate is studied [4]. A future computer system must be capable of performing 100 Exaflops [2] (10^{18}) to 10 Zettaflops (10^{21}) [38].

In nanoscience, a grid-enabled version of the Fragment Molecular Orbital (FMO) method for Petascale Computing was developed. The FMO method can execute all electron calculation in large molecules with more than $10s$ of thousands of atoms. To execute the FMO program a Petascale computer system with 10 Petaflops is needed while it was simulated on a 0.5 Petaflops computer system [104].

High-performance computing will allow astrophysicists to investigate astrophysical objects, systems, and events that cannot be studied by available computer systems [79]. The total required computations for Gamma-Ray Bursts (GRBs)[3]

1-Global climate models (GCMs) are a class of computer-driven models for weather forecasting, understanding climate and projecting climate change.

2- flops (or FLOPS or flop/s) is an acronym meaning FLoating point Operations Per Second.

3-Gamma-ray bursts (GRBs) are flashes of gamma rays associated with extremely energetic explosions in distant galaxies.

simulation is 270 Exaflop. Consequently, a computer system capable to run at 1 Petaflop/s performance would require 3 days [79].

1.1.2 High-performance Computers: Current and Future

In this section, the current-day systems and a projection to show the performance needed of future computers are presented.

The Berkeley Open Infrastructure for Network Computing (BOINC) is a non-commercial middleware system for grid computing [17]. BOINC has about $570,000$ active computers (hosts) worldwide processing on average 1.9 Petaflops as of July 2009. NASA, Intel and SGI have announced to build a 1 Petaflops computer "Pleoades" in 2009 that expect to reach 10 Petaflops until 2012 [3]. At the same time, IBM intended to build a 20 Petaflops supercomputer "Sequoia" at Lawrence Livermore National Laboratory until 2011. With the current speed of progress, Supercomputers are projected to reach 1 Exaflops in 2019. E. P. DeBenedictis of Sandia National Laboratories theorizes that a Zettaflop computer is required to accomplish full weather modeling, which could cover a two week time span accurately [1]. Such systems might be built around 2030. In the following the projection done in TOP500 project presents the future of high-performance computers and their performance. The TOP500 project was launched in 1993 to provide a reliable basis for tracking and detecting trends in high-performance computing [73], [74]. Twice a year, a list of the sites operating the worlds 500 most powerful computer systems is compiled and released. The list contains a variety of information including the systems specifications and major application areas. The projection into the future, based on 30 lists of real data, by a least square fit[1] on the logarithmic scale is depicted in Figure 1.1.

From Figure 1.1, it can be observed that, in 2015, it is expected there will be only Petaflop/s systems in the TOP500 list. The projection also shows that the first Exaflop/s computer will enter the TOP500 list in 2019, and only one year later, in 2020, there will be the first notebooks with a performance of 100 Teraflop/s. The TOP500 project provides a reliable basis for tracking and detecting trends in high-performance computing.

1-A mathematical procedure to find the best-fitting curve to a given set of points by minimizing the sum of the squares.

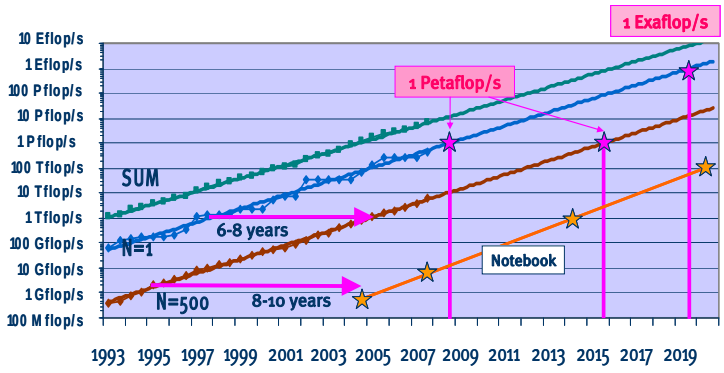


Figure 1.1: Performance projection for computing systems. The curves show summation, position 1 (N=1), position 500 (N=500), and notebook levels performance (Source: Meuer [74]).

1.1.3 High-performance Processing: Resources and Requirements

As discussed earlier, many applications need a high-performance processing system to execute efficiently. High-performance processing stems from the provided raw performance and the efficient exploitation parallelism within a computing system [22]. A well-known technique to achieve high performance is to execute as many as possible parts of an application in parallel. Supercomputers are a good candidate to execute the high-performance applications. Some concerns limits the utilization of supercomputers. The biggest issue is related to their cost that makes them unaffordable for some applications. Another issues with running supercomputer are cooling and the speed at which information can be transferred or written to a storage device. The speed of data transfer will limit the performance of supercomputer. Supercomputers consume and produce massive amounts of data in a very short period of time. A supercomputer turns compute-bound problems into I/O-bound problems, hence, much work on external storage bandwidth is needed to ensure that this information can be transferred quickly and stored/retrieved correctly. Supercomputers get extremely hot when they run. Therefore, they require complex cooling systems to ensure that no part of the computer fails.

An effective solution to achieve the high-performance is the utilization of grid

computing. The exploitation of the parallelism available in grid computing is a very attractive solution to execute either different parts of an application or several applications at the same time. The main advantage of grid computing is that each node can be purchased as a simple computer system, e.g., desktop computer or server, which when combined can produce similar computing resources to a multiprocessor supercomputer, but at a lower cost. The most important capabilities of grid computing are as follows [24], [69]: exploiting under-utilized resources, parallel processing capacity, and reliability. A grid system provides some mechanisms to exploit unused resources, e.g., desktop machines, and servers. It manages and resubmits tasks when a failure is detected to provide a reliable grid system. In a grid system, the most important resources are the **computing** and **communication** resources [24]. The computing resources are provided by the processors of the machines on the grid that these machines connected to a network (private, public, or the Internet).

The rapid growth in communication bandwidth today makes grid computing practical, compared to the limited bandwidth available when distributed computing first emerged. With the increase in improved communication more applications can benefit from grid. It should be noted that without adequate communication bandwidth, grid computing would be not possible. Communications bandwidth within the grid are important for sending tasks and their required data to points within the grid. Some tasks require a large amount of data to be processed which may not always reside on the machine running the task. The bandwidth available for such communication can often be a critical resource that can limit utilization of the grid.

Increasing the bandwidth in a grid node needs to exploit the suitable network processing equipments. Within a grid node network processing operates as an integral part to accelerate the processing of packets. Network processing refers to the tasks performed on packets by network equipments. The important part of network processing equipments are mainly routers and switches [9], [70], [115]. These networking equipments are comprised mainly of network elements that are called network processors (NPs). Typically, a network processor comprises a parallel programmable processor core with a number of memory interfaces and special co-processors that are optimized for packet processing [9], [88], [112]. Network processors combine the flexibility of general-purpose processors with the high-performance of application-specific integrated circuits (ASICs).

The performance of these network processors as main part of the network processing to achieve higher bandwidth is usually hampered by slow memory accesses. Such memory bottlenecks can be overcome by the following mecha-

nisms: hiding of memory latencies through parallel processing and reducing the memory latencies by introducing a multi-level memory hierarchy incorporating special-purpose caches. An approach to achieve higher lookup performance is to utilize the Bloom filter data structure that recently is utilized by network processors [13], [39]. A Bloom filter is a simple space efficient randomized data structure to represent a set in order to support membership queries [25]. A Bloom filter is frequently utilized in network processing (areas), such as packet classification, packet inspection, forwarding, peer-to-peer (P2P) networks, and distributed web caching [27], [30], [39].

Not only performance but also flexibility is an important factor in designing a grid-based high-performance computing systems. In order to design a flexible grid-based high-performance computing system, it must utilize the processor architectures with sufficient flexibility as well as high-performance. Different approaches such as General-Purpose Processors (GPPs) and application-specific processors have been proposed to provide flexibility and high-performance. GPPs have been widely used in the past decades and they provide high degree of flexibility. Application-specific processors provide more performance than GPPs for specific applications they are designed to accelerate, while their flexibility is limited. In order to provide the flexibility of GPPs with the high-performance of application-specific processors, reconfigurable processors are utilized. Reconfigurable computing has proven to be a promising technology to increase the performance of certain algorithms in scientific and engineering applications in recent years. Any application of iterative nature such as multimedia processing, digital signal processing, bioinformatics, cryptography and software defined radio, etc. can be mapped on an FPGA by programming it using hardware description languages (HDLs). These applications have certain kernels containing iterations which are processed in parallel on the processing elements on an FPGA. The same applications can take much longer time when they are run on a general-purpose processor (GPP) that processes the iterative kernels in a sequential manner [26], [36], [113]. Therefore, the use of reconfigurable processors should be beneficial to achieve high-performance computing in a grid system for compute-intensive applications. The target applications are computationally intensive applications. When these applications are executed on a grid node with the reconfigurable processor, the reconfiguration time is very small as compared to computation time. Consequently, focusing on the reconfigurable processors as a computing resource and Bloom filters for the packet processing in network processing applications provides the performance and flexibility required for a high-performance computing system.

1.2 Challenges and Goals

In the previous section, we argued that flexibility and performance are two important factors in the design of a high-performance computing system. Reconfigurable processors provide flexibility and performance for computationally intensive applications. Furthermore, accelerating the network processing tasks for the existing computing resource in grid computing makes it more beneficial. Consequently, our general goal is to investigate:

- How to achieve high-performance and flexible processing in networked and grid environments?

It is not our intention to propose an all-encompassing methodology for the design of grids to support all (current and future) applications. However, we identified that packet processing is a recurring operation and an integral part of all grids. Consequently, we will focus on how to speed up packet processing to further improve the (communication) performance of grids. In particular, we identified that the Bloom filter is a promising technique to accelerate packet processing. Our first problem statement is:

- How can Bloom filters be employed to speed up network processing in the communication side of the grid nodes?

Bloom filters have been utilized in many applications to quickly determine the membership of elements within a large dataset. The applicability of rules on incoming packets can be perceived as a similar operation and, therefore we expect that Bloom filters can contribute to the speedup of packet processing. This is because the memory bottleneck due to the gap between memory and processor in the packet processing systems can be overcome by utilization of Bloom filters. The used Bloom filters for packet processing is exploited as an integral part of the grid node.

As said before, flexibility in grids is a requirements that is gaining much attention. We believe that reconfigurable computing could possibly be the solution to provide the needed flexibility and at the same time improve the performance of computing nodes. Our second problem statement is:

- How can reconfigurable computing be utilized to improve flexibility and performance in grids?

Reconfigurable computing has proven itself to be able to speed up many applications despite its lack in achieving high frequencies. However, frequency is not the sole factor that determines performance. Field-programmable gate arrays (FPGAs) - as the most utilized reconfigurable fabric nowadays - provide a large amount of parallel structures that when exploited efficiently can greatly contribute to the speedup of applications. It is expected that the parallelism in many applications can be exploited to achieve higher performance. This would immediately introduce two types of parallelism in our envisioned new grid environment - coarse-grain parallelism and fine-grain parallelism. Applications running in grids can be broken down into (coarse-grain) tasks that are distributed over the grid nodes, and (fine-grain) sub-tasks can be distributed over the parallel processing elements within a single grid node. This distinction opens up the (physical) possibility for collaborative grid computing as grid nodes can now decide "how much parallelism they can handle" and forward other tasks (tasks or sub-tasks) to other collaborating grid nodes. Consequently, we define several goals within the second problem statement:

- Propose an initial organization of grid nodes encompassing reconfigurable processors
- Determine the performance gains of utilizing reconfigurable processor within grid environments

The reconfigurable processors used in grid nodes execute the submitted tasks. It should be noted that the submitted tasks should have the inherent parallelism to achieve more performance. The level of parallelism accepted by each node is depended on different factors such as: application structure, and the grid node specifications.

1.3 Methodology

In this section, we propose the different steps to achieve a high-performance processing system as proposed in the previous section. These steps are listed in the following:

- Investigate the collaboration between reconfigurable processors (elements) in a grid computing environment
- Propose several approaches to achieve more performance in network processing applications using Bloom filters

- Introduce an optimal bandwidth allocation approach for network processors in the networked and grid environments using queueing theory

We propose the collaboration of reconfigurable processors in grid computing. This approach provides a flexible and high-performance framework to execute computationally intensive applications. A combination of both reconfigurable processors and grid computing is known as Collaborative Reconfigurable Grid Computing. Figure 1.2 depicts a general overview of high-performance processing environments with the place of grid computing.

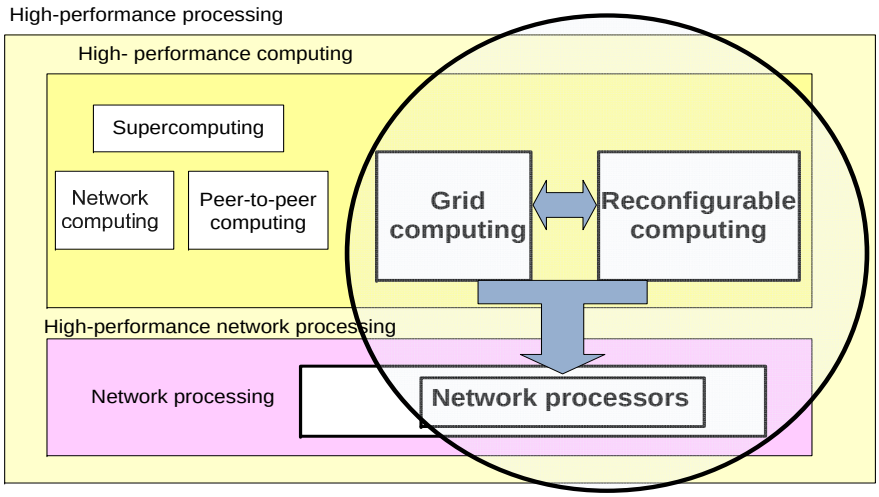


Figure 1.2: General overview of the high-performance processing environment on compassing grid computing, reconfigurable computing and network processing .

In Figure 1.2, the circle shows that we focus on grid computing, reconfigurable computing, and the required network processing. To achieve higher performance in the network processing systems, we focus on Bloom filters. Indeed, due to its wide applicability in network processing, some modern network processors incorporate a Bloom filter unit in their implementation. There are numerous networking problems where such a data structure is required. Especially when space is an issue, a Bloom filter may be an excellent alternative to keep an explicit list of the items. The major types of Bloom filters are the following: standard Bloom filter, compressed Bloom filter, counting Bloom filter, distance-sensitive Bloom filter, Bloom filter with two hash functions,

space-code Bloom filter, spectral Bloom filter and dynamic Bloom filter. These different Bloom filters are utilized in network processing and database applications [27], [39], [49]. We propose three additional types of Bloom filter to achieve more performance in network processing:

- A multi-level memory architecture for a counting Bloom filter (cached counting Bloom filter)
- A memory optimization approach for Bloom filters using an additional hashing function
- A k-stage pipelined Bloom filter to decrease power consumption

These approaches are implemented for hash-based packet classification. The advantages of these approaches are:

- Overcome memory bottleneck in network processors and packet processing applications
- Increase the performance of network processing applications

Finally, we investigate an optimal bandwidth allocation approach for network processors in the networked and grid environments using queueing theory and Jackson models. In this approach to minimize the response time, the optimal arrival rate for different NPs in a grid-oriented environment is determined.

1.4 Thesis Overview

This section discusses the organization of the remainder of this dissertation which consists of the following chapters:

- Chapter 2 introduces some basic concepts and the necessary background to better understand the remainder of this dissertation. It introduces the definition of grid computing, reconfigurable processors, and the concept of network processors and Bloom filter. Moreover, the performance analysis of network processors using queuing theory is presented.
- Chapter 3 highlights the concept of collaboration of the reconfigurable processors in a grid environment. It also describes the proposed solution applied to multimedia kernels.

- Chapter 4 presents the Bloom filters in high-performance network processing. It introduces the details of the three proposed approaches for Bloom filters which include a cache organization for counting Bloom filter, memory optimization approach for Bloom filter using an additional hashing function, and a pipelined Bloom filter for packet classification.
- Chapter 5 describes the performance evaluation model and optimal bandwidth allocation for network processing systems.
- Chapter 6 presents the experimental results of the proposed solutions for high-performance processing that were discussed in the proceeding chapters. It analyzes the results to demonstrate the benefits of the proposed solutions.
- Chapter 7 presents the conclusion of this dissertation and describes the main contributions of the described research. Finally, several future research directions to continue the described work are presented.

Chapter 2

High-performance Processing Background

Advancement *of science and engineering and their impact on large scale applications can be related to the progress and availability of high-performance processing systems and grids. Each high-performance processing system comprises computing and communication resources. Traditionally, general-purpose processors are utilized inside the grid as computing resources, which can be replaced by reconfigurable processors to provide more flexibility and performance. In addition, it must obvious that computing resources need a high-performance network processing within the communication resources. The network processing is investigated as major part of communication resources in a high-performance processing system. Network processing refers to the tasks performed on packets by network equipment, e.g., routers and switches. These networking equipments are comprised mainly of network elements that are called network processors (NPs).*

This chapter presents a background in the area of the high-performance computing systems, and the network processing. Section 2.1 presents two high-performance computing systems: grid computing and reconfigurable computing. Section 2.2 introduces the network processing and applications. Section 2.3 presents related work. Finally, Section 2.4 summarizes this chapter.

2.1 High-performance Computing Systems

In this section, we briefly present grid computing and reconfigurable computing. We discuss the motivation for grid computing, its capabilities, the concept of reconfigurable processors and their benefits.

2.1.1 Grid Computing

A well-known method to increase the performance of large scale applications with inherent level of parallelism is to execute as many as possible parts of the application in parallel. The exploitation of the parallelism available in grid computing is a very attractive solution to execute different part of an applications. This is because, computing system e.g., desktop computers, servers in the grid can be used as computing resource with lower cost in compared to the supercomputers. Grid computing has emerged as the next generation parallel and distributed computing methodology, which aggregates dispersed heterogeneous resources for solving various kinds of large-scale parallel applications in science, engineering, and commerce. Grid computing, peer-to-peer computing and traditional network computing can all be considered to be part of the distributed computing context. Grid computing is defined in literature as systems and applications that integrate and manage resources and services distributed across multiple control domains[1] [33].

Some resources may be used by all users of the grid while others may have specific restrictions [69]. Figure 2.1 depicts the place that grid computing occupies within the distributed computing environment.

Grid computing is similar in structure to standard network computing and peer-to-peer (P2P). In standard network computing, users are members of a single organization and a network administrator has access to all of them. In grid computing, users/resource owners are members of many organizations, or may be individual private elements. There are three main types of grids as follows:

- Computational grids: the computational grid focuses on dedicating resources for computing power; i.e., solving equations and complex mathematical problems. Machines participating in this type of grid are usually high-performance servers [33].
- Data grids: the grid architecture is responsible for storage and providing access to large volumes of data, often across several organizations.

1- A collection of resources controlled by a single support staff.

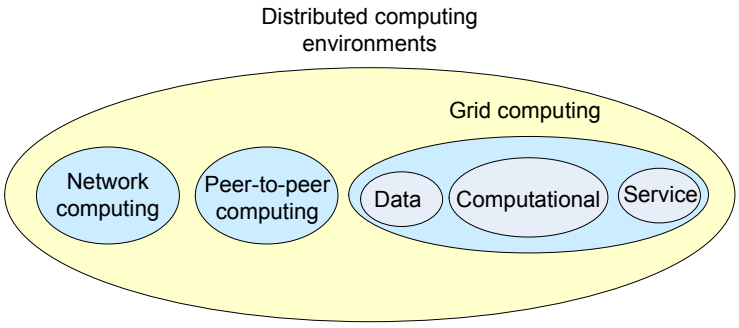


Figure 2.1: General overview of the distributed computing environment with the place of grid computing.

Storage can be memory attached to the processor or it can be secondary storage using hard disk drives or other permanent storage media. Memory attached to a processor usually has very fast access but is volatile. Secondary storage in a grid can be used in interesting ways to increase capacity, performance, sharing, and reliability of data.

- Service grids: the grid provides services that are not available on a single grid node [33]. Service grid is a system that provides function, program licence, resource and support dynamic creating, running, maintaining, and canceling of applications [69].

Grid Capabilities

The most important capabilities of grid computing are as follows [33], [69]:

- Exploiting under-utilized resources: in most organizations, there are large amounts of under-utilized computing resources. Most desktop machines are busy less than 5% of the time. In some organizations, even the server machines can often be relatively idle. Grid computing provides a framework for exploiting these under-utilized resources, therefore has the possibility of substantially increasing the efficiency of resource usage.
- Parallel CPU capacity: the potential for massive parallel CPU capacity is one of the most attractive features of a grid. The common attribute among such uses is that the applications have been written to use algorithms that can be partitioned into independently running parts. A computationally

intensive grid application can be thought of as many smaller sub-tasks, each executing on a different machine in the grid.

- Virtual resources and virtual organizations for collaboration: the users of the grid can be organized dynamically into a number of virtual organizations, each with different policy requirements. These virtual organizations can share their resources collectively as a larger grid.
- Access to additional resources: In addition to CPU and storage resources, a grid can provide access to increased quantities of other resources and to special equipment, software, licenses, and other services.
- Resource balancing: for the grid-enabled applications the grid can offer a resource balancing effect by scheduling grid tasks on machines with low utilization.
- Reliability: grid management software can automatically resubmit tasks to other machines on the grid when a failure is detected. In critical real-time situations, multiple copies of the important tasks can be run on different machines throughout the grid.

2.1.2 Reconfigurable Processors

Reconfigurable computing is defined as the study of computation using reconfigurable devices [26]. Configuration and reconfiguration is the process of changing the structure of a reconfigurable device at the time of start-up and at run-time, respectively. Reconfigurable devices, including field-programmable gate arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements are known as logic blocks and are connected using a set of routing resources that are also programmable [26].

It is possible to describe a design simply by writing logical expressions, a level higher than gates. Register transfer level (RTL) design is a popular discipline for describing these logical expressions. It allows the designer to express the design by describing the logic between each pair of register stages. VHDL is one popular programming language that supports RTL hardware descriptions [54]. VHDL is a strongly typed, Ada-based programming language that includes special constructs and semantics for describing concurrency at the hardware level. These concurrency constructs are new for most programmers and can be a source of confusion for beginners. VHDL is not a case sensitive language.

One can design hardware in a VHDL IDE (such as Xilinx) to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software (such as ModelSim) which shows the waveforms of inputs and outputs of the circuit after generating the appropriate testbench. To generate an appropriate testbench for a particular circuit or VHDL code, the inputs have to be defined correctly. The key advantage of VHDL when used for systems design is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires). Another benefit is that VHDL allows the description of a concurrent system (many parts, each with its own sub-behavior, working together at the same time). VHDL is a Dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

Hardware/software partitioning, is the process of dividing an application between a microprocessor component (software) and one or more custom coprocessor components (hardware) to achieve an implementation that best satisfies requirements of performance, size, designer effort, and other metrics. Hardware/software partitioning is a hard problem because of the large number of possible partitions. In its simplest form, hardware/software partitioning considers an application as comprising a set of regions and maps each region to either software or hardware such that some cost criteria (e.g., performance) is optimized while some constraints (e.g., size) are satisfied. A partition is a complete mapping of every region to either hardware or software. Even in this simple formulation, the number of possible partitions can be enormous. If there are n regions and there are two choices (software or hardware) for each one, then there are 2^n possible partitions.

Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. If we consider two scales, one for the performance and the other for the flexibility, then the general-purpose computers can be placed at one end and the ASIPs[1] at the other end as depicted in Figure 2.2.

Between the GPPs and the ASIPs are reconfigurable processors. GPP is more flexible than reconfigurable processor because of the higher design effort entailed for designing in hardware. Ideally, we would like to have the flexibility

1-A processor designed for only a single application is called an Application-Specific Instruction-set Processor (ASIP). The instruction set of the application is directly implemented in hardware.

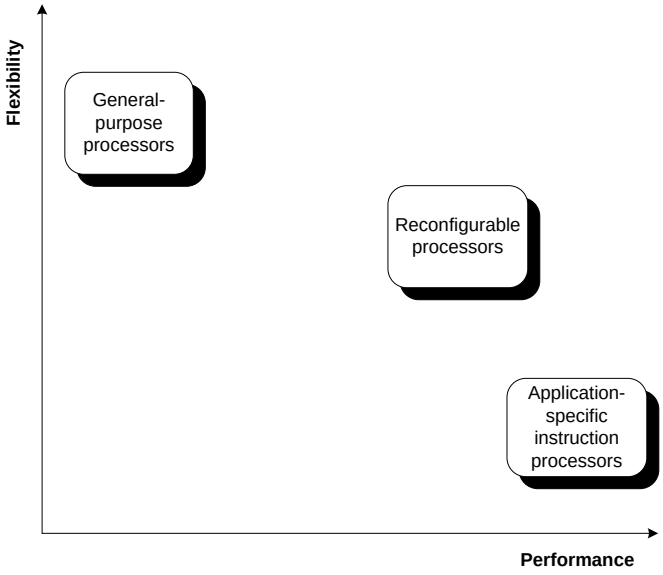


Figure 2.2: Performance versus flexibility in different architectures.

of the GPP and the performance of the ASIP in the same device. We would like to have a device able to adapt to the application quickly. Therefore, a reconfigurable processors can be such device to achieve both performance and flexibility [26].

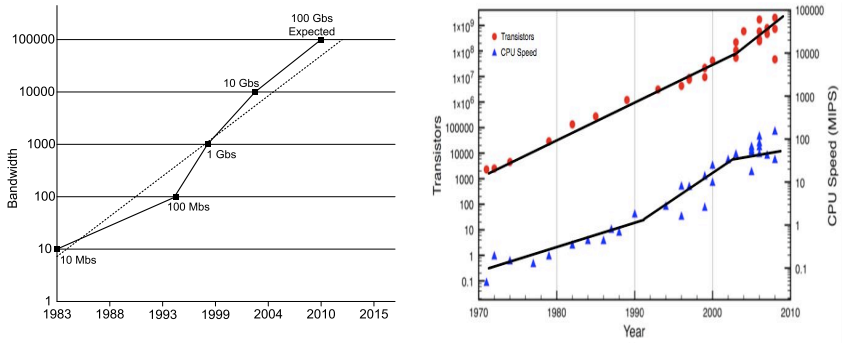
2.2 High-performance Network Processing

In this section, the network processing concept and performance modeling of network processing using queuing theory is presented. Subsequently, packet classification and Bloom filters as an instance application and solution in the network processing are highlighted.

2.2.1 Network Processing

The bandwidth growth of networks increased almost exponentially in the recent years and is expected to continue to do so for years to come. This has been fueled by emerging new technologies that are capable of achieving higher

bandwidths. Consequently, new applications are being developed that take advantage of the new capabilities. In turn, more consumers are starting to use these applications and thereby further increasing the demand for higher bandwidth. As networks become the infrastructure for information, interactive data, real-time data, huge multimedia content transport, and many other services, the technology of networks must cope with various requirements, but primarily that of speed. High-speed networking refers to two aspects of speed: the links transmission rates from multi-Mbps (10^6 bits per second) to multi-Gbps (10^9 bits per second) and the complexity and speed of the required processing due to the number of networks, addresses, services, traffic flows, and so on. If we examine the speed of network links over the years, we find a similar but a higher growth pattern than that of processing capabilities. In computing, this growth is considered to double after every 2 years (according to Moores law[1]). For example, If we look at Ethernet bandwidths, we find a speedup of 10^4 in the past 27 years (from 10 Mbps approved in 1983 to 100 Gbps expected to be approved in 2010, as depicted in Figure 2.3) which is doubling the bandwidth every 24 months [35].



(a) (b)

Figure 2.3: (a) Bandwidth growth on the Ethernet interface (the dot-line shows the projection of bandwidth growth). (b) Transistor and MIPS (millions of instructions per second) trends over time (source [75]).

However, if we examine the increase from 100 Mbps (approved in 1995) to 100 Gbps, it is doubling the bandwidth every 18 months. Ethernet bandwidth growth is going according to Moore's law. The technological advances must also be

1-Moore's law: The prediction by Gordon Moore that the number of transistors on a microprocessor would double periodically (approximately every two years).

accompanied by improved network processing capabilities within routers and switches that connect the networks. Therefore, network processors have been incorporated within these devices to cope with the continuously increasing demand for higher performance. Consequently, the design of network processors remains an ongoing research and development effort. A network processor comprises a parallel programmable processor core with a number of memory interfaces and special co-processors that are optimized for packet processing [9], [88], [112]. The network processor differs from traditional microprocessors in two ways:

1. The instruction set of an NP contains special instructions for particular operations, e.g., bit manipulation, CRC calculation, and search and lookup operations.

2. Special hardware function blocks are present to accelerate specific packet processing tasks.

Finally, a network processor can be utilized in two different planes that differ in the speed and manner they handle incoming packets; namely data plane and control plane. This model is depicted in Figure 2.4.

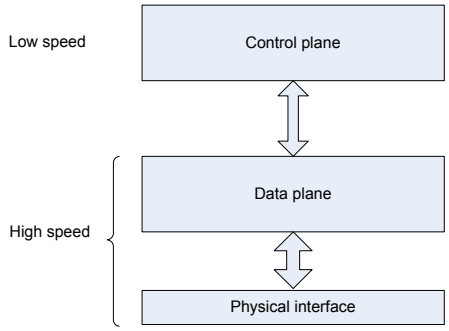


Figure 2.4: Packet processing model in network processor.

In the data plane simple tasks are performed, and most packets follow the fast path through the NP that required very little processing. In the control plane exceptional packets and complex routines are handled [46]. Fast path packets correspond to data plane tasks, while slow path packets correspond to control plane tasks.

Network processor general requirements

The most important requirements in the network processor systems are given in the following:

- **Performance:** By executing key computational kernels in hardware, NPs must be able to perform many applications at wire speed. Network processors must be able to support high bandwidth connections, multiple protocols, and advanced features without becoming a performance bottleneck.
- **Programmability:** Having software as a major part of the system allows network equipment to easily adapt to changing standards and applications. The network processor should be easily programmable in order to support customization of feature sets and the rapid integration of new and existing technologies. In order to meet this demand, network processor manufacturers must strive to supply programming and testing tools that are easy to use. These programming tools should be based on a simple programming language that allows for reuse of code wherever possible. In addition, programming tools must provide extensive testing capabilities that provide intelligent debugging features, such as descriptive codes and definitions, as well as code level statistics for optimization [88].
- **Fast time to market (TTM):** Time to market has become a critical factor in achieving success with network equipment, it is the time required for system vendor to bring a product from demand to commercial availability and is known as a factor that determined the success or failure of the product in the market [32].
- **Serviceability:** Users are demanding services such as real-time video, secure private networks and voice over IP, these will require lot of serviceability at the access and edge network elements [9].

Network processor functionalities

Typical functions performed by network processors are summarized below:

- **Lookup and pattern matching:** This function compares packet header fields with specific patterns to classify the type of packets, for example it performs a table lookup to return the relevant table entry or determine the type of incoming packets, whether it is an IPv4 or an IPv6 packet.

- **Forwarding:** This function is defined as determining the output path for incoming packets. It is implemented using hardware prefix tree structure and special hardware [2].
- **Access control and queue management:** Once packets are identified, they are placed in appropriate queues for further processing. Packets are also checked against security access policy rules to see if they should be forwarded or discarded.
- **Traffic shaping and control:** Some protocols or applications require that, as traffic is released to the outgoing wire or fiber, it is shaped to ensure that it meets delay or delay variation requirements. Other requirements specify the priority of traffic between different channels or message types [102].
- **Data manipulation:** This is where the packet is modified in some way, this could be decrementing the Time To Live (TTL) field in a IP packet, recalculating the CRC check, performing packet segmentation and reassembly and encryption or decryption of packets.

Network processor implementation

Each network processor is combination of many different elements, that are described by the following:

- **The processing engine:** The basic programmable unit in the network processor is a processing engine (PE). The PE may be clustered in a group of two or more PEs. Different network processor use different architectures for their PEs. The PEs may be grouped into functional blocks or can be independent. Moreover, next to network processors sometimes co-processors or hardware accelerators are utilized. A hardware accelerator is the finite state machine that operates independently of the PEs and is called a functional unit. If a hardware accelerator is programmable it is called a co-processor. The abstract model of network processor is depicted in Figure 2.5.
- **Exploiting parallelism:** All network processors are using parallel techniques and pipelining. Basically, They use three types of parallelism:
  1. Instruction-level parallelism (ILP).
  2. Thread-level parallelism (TLP).

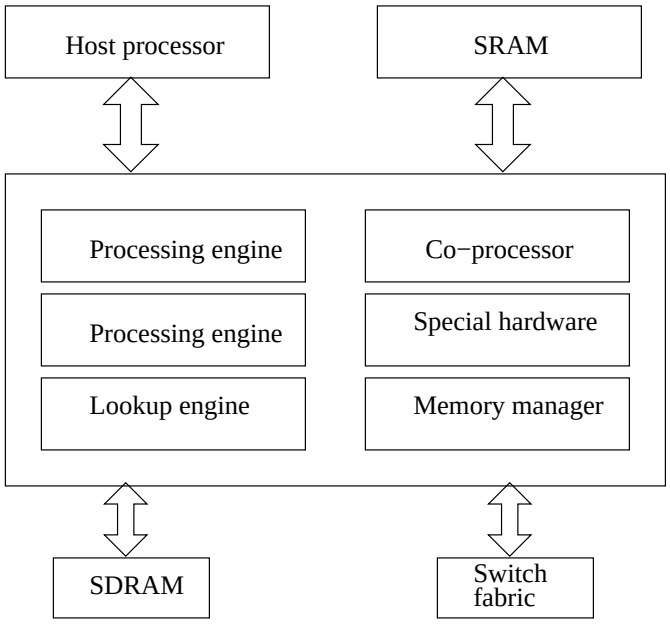


Figure 2.5: Abstract model of network processor organization.

3. Packet-level parallelism (PLP).

In ILP, the compiler or hardware instruction scheduler determines simultaneous execution of program instructions. In TLP, different threads are executed to avoid idle time in memory references and processing engines, i.e., if a thread waits for the memory it is stalled and then another thread is started. In PLP, a mechanism should be used for packet ordering to allow parallel processing of packets.

- **NP memory organization:** A critical resource in NPs is the memory organization. There are three types of memories in NPs which include: instruction memory, packet memory, and route table memory. Instruction memory is usually small because the number of instructions in NP is low. Packet memory that handles the buffered arrival packets, queued, modified packets and read forwarding packets must be designed carefully with a minimum delay. Routing table memory includes routing entry that is read by the NP. The routing table requires update operations and lookups thus it must be designed as fast as possible. One solution for

mentioned aim is use intelligent data structure, hardware accelerator for lookup, content addressable memory (CAM) and SRAM.

- **Dedicated hardware:** All network processors incorporate special hardware and integrated co-processors to perform common networking tasks. Typical hardware functionality include CRC calculation, queue management, forwarding engine and lookup engine.
- **Network interface:** The most important feature next to a network processor is the network interface. This is the point where packet enters and exit the network processor.

2.2.2 Performance Modeling using Queuing Theory

Queuing analysis is an important tool in computer and network modeling. A queue is defined as a collection of items in which only the earliest added item is accessed. Basic operations are add/enqueue (to the tail) and delete/dequeue (from the head). The delete operation returns the item removed. Queuing is the method by which tasks are ordered to access a computer resource. The basic queuing model is depicted in Figure 2.6. It can be used to model computer systems or communication equipments [5].

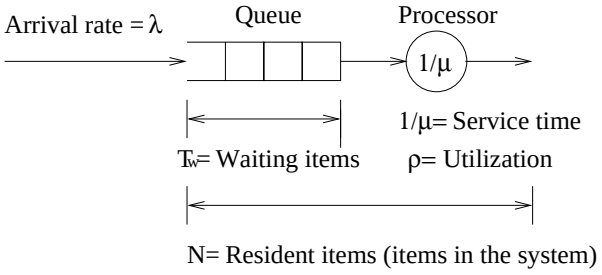


Figure 2.6: Simple queuing model.

In Figure 2.6, λ, μ and ρ show arrival rate, service rate and utilization, respectively. There are different kinds of queuing models as follows:

1. Single-server queues
2. Multiple-server queues
3. Network of queues

The single-server queue is the simplest queuing model in that the central element of the model is a single server that provides some service to input items. If the server is idle, an item is served immediately. Otherwise, an arriving item should wait. When the server completes serving an item, the item departs. If there are items waiting in the queue, the next one is immediately dispatched to the server (see Figure 2.6). The multi-server queue is a generalization of the single server queue where all servers share a common queue. It is assumed that all servers are identical; thus, if more than one server is available, it makes no difference which server is chosen for the item. Whenever a server is ready to serve and items still reside in the queue, the item at the front of the queue is dispatched to the respective server. If all servers are busy, a queue begins to form. The multi-server queue model is depicted in Figure 2.7.

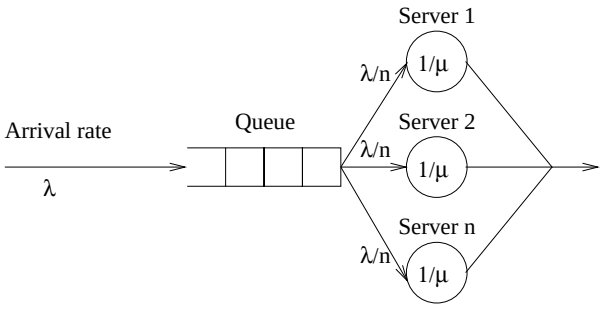


Figure 2.7: Multi-server queuing model.

Kendall introduced a simple notation to characterize queuing models [5], [21], [99]. It is a three-part code denoted as $a/b/c$. The first letter specifies the inter-arrival time distribution, the second one represents the service time distribution and the last letter specifies the number of servers. Some examples are: $M/M/1$, $M/M/c$, $M/G/1$, $G/M/1$ and $M/D/1$, where the letter G shows a general distribution, M for the exponential distribution (M stands for memoryless property of exponential distribution), and D for deterministic times [5].

Networks of queues and the Jackson theorem

Queuing network analysis is a valuable tool in determining the performance and operating characteristics of real-world networked systems. A queuing network is a collection of two or more nodes where items are being serviced. Items arriving at the network request service from one or more of the nodes and

then may leave the network [21]. A fundamental and simple characteristic of queuing networks is whether they are open or closed. An open network allows items to enter and leave the network. In a closed network, items are "trapped" and circulate among the various nodes in the network.

An example of open and close queuing models is depicted in Figure 2.8 The dashed box in the figures indicates the logical boundary of the queuing network. The circles are the nodes where items receive service. The arrows indicate the paths items may take within the network.

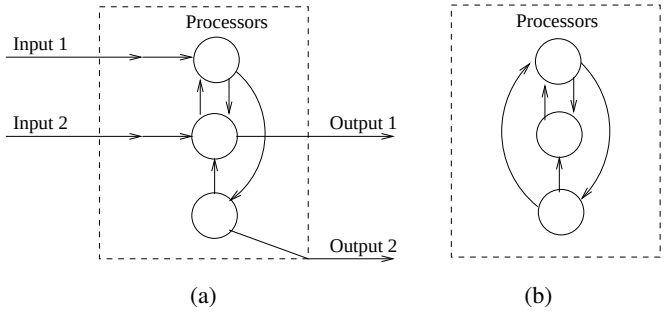


Figure 2.8: Closed and open queuing networks. (a) An open network. (b) A closed network.

Classification of queueing networks is especially important. Many classes of networks have no known closed-form solutions. A network might contain different classes of input items. An example of this can be found in computer systems where user jobs enter and exit the system but certain system-level jobs are always present and circulate continuously within the system. This is a mixed network. There are several analysis methods related to queuing models. These methods have been divided into two types: exact and approximate [21]. The exact method, means a solution that is exact with respect to the assumptions. Approximate method means that the solution, more or less, corresponds to what occurs in a (presumably) more accurate model of a network. One exact model to analyze queuing networks is called Jackson network. Jackson model for open queuing networks is normally used when the incoming tasks are of the same task class. A Jackson network consists of M nodes that satisfy the following conditions:

1. Each node consists of c_i identical exponential servers (the server with exponential service time distribution) where the service rate of the i_{th} node is μ_i.

2. Items arrive from outside the system to the i_{th} node according to a Poisson process[1] with rate s_i. Items may also arrive from other nodes within the network.

3. Items from node i are routed to node j with probability p_{ij} or leave the network with probability $1 - \sum_{j=1}^{M} p_{ij}$.

The arrival rate λ_i to each node i from all sources (external and internal) is

$$\lambda_i = s_i + \sum_{j=1}^{M} p_{ji}\lambda_j,\ i = 1, ..., M \tag{2.1}$$

In this equation, s_i is the external arrival rate in each node, p_{ji} is the routing probability between node j and node i, λ_j is arrival rate to node j. For each network, we have M arrival equations and these equations form a solvable linear system. For networks that satisfy the above conditions, Jackson proved that networks can be described by a $M/M/c_i$ model with arrival rate λ_i and service rate μ_i. An open Jackson network example is depicted in Figure 2.9.

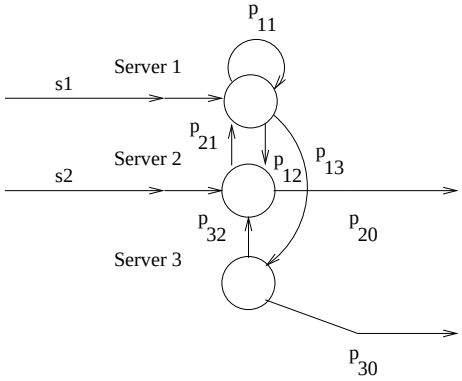


Figure 2.9: Open Jackson network.

We analyze it to find the network parameters. The arrival and service rate for the servers and routing probabilities (that node 0 is assumed outside the network) are: $p_{11} = 0.05$, $p_{12} = 0.45$, $p_{13} = 0.5$, $p_{21} = 0.1$, $p_{20} = 0.9$, $p_{30} = 0.8$,

1-Poisson process is an important stochastic process used in computer systems performance evaluation. A Poisson stochastic process has the property that events are independent, and the inter-arrival times of events can be described using the exponential distribution [47].

$p_{32} = 0.2$ $s_1 = 10^6$ $item/sec$, $s_2 = 2 \times 10^6$ $item/sec$, $\mu_1 = 1.5 \times 10^6$ $item/sec$, $\mu_2 = 3 \times 10^6$ $item/sec$, $\mu_3 = 10^6$ $item/sec$.

Using Equation (2.1), the arrival equation for each server is:

$$\begin{aligned} \lambda_1 &= s_1 + 0.1\lambda_2 + 0.05\lambda_1 \\ \lambda_2 &= s_2 + 0.45\lambda_1 + 0.2\lambda_3 \\ \lambda_3 &= 0.5\lambda_1 \end{aligned} \tag{2.2}$$

Solving these linear equations in Equation (2.2) yields $\lambda_1 = 1327433\, item/sec$, $\lambda_2 = 2610619$ $item/sec$, $\lambda_3 = 663717$ $item/sec$. In the Jackson network [53], [114], important parameters are: the mean number of items (mean queue length) and mean resident time in network (response time). The mean number of items in each node i (N_i) with utilization ρ_i is: $N_i = \dfrac{\rho_i}{1-\rho_i}$. Therefore, the total mean number of items in the network is:

$$\bar{N} = \sum_{i=1}^{M} N_i = \sum_{i=1}^{M} \frac{\rho_i}{1-\rho_i} \tag{2.3}$$

The mean resident time (response time) in the network of an item is:

$$T_s = \frac{\bar{N}}{\lambda} = \frac{1}{\lambda}\sum_{i=1}^{M} \frac{\rho_i}{1-\rho_i} = \frac{1}{\lambda}\sum_{i=1}^{M} \frac{\lambda_i}{\mu_i - \lambda_i} \tag{2.4}$$

In the Jackson network for optimal capacity allocation, we assume to have control over the service rates $\mu_1, ..., \mu_M$ with the constraint that the total service capability is fixed to a constant value c as follows: $c = \sum_{i=1}^{M} \mu_i$. For a given set of arrival rates λ_i, the optimal set μ_i that minimizes the average number of items in the network $\bar{N} = \sum_{i=1}^{M} N_i$ is [53]:

$$\mu_i = \lambda_i + \frac{\sqrt{\lambda_i}}{\sum_{i=1}^{M}\sqrt{\lambda_i}}(c - \sum_{j=1}^{M} \lambda_j) \tag{2.5}$$

Jackson model for open queuing networks is normally used when the incoming tasks are of the same task class with networks of single-server queues having exponentially distributed service times.

2.2.3 Packet Classification

Traditionally, routers forward packets based on the destination address in the packet. The support of many different services such as Quality of Service (QoS),

Virtual Private Network (VPN), policy-based routing, traffic shaping, firewalls, and network security, increases the importance of packet classification. In order to provide these services, the router must categorize the incoming packets according to different criteria. These criteria are determined based on one or more fields in the packet header. Packet header fields include destination and source IP addresses, the protocol type, and the destination and source port numbers is depicted in Figure 2.10.

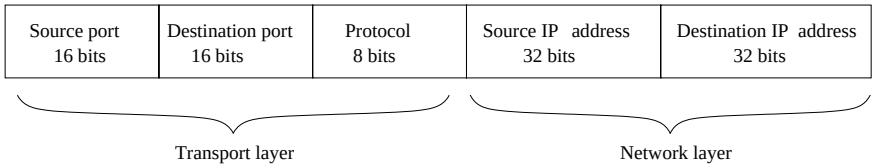


Figure 2.10: Important fields that are used in packet classification algorithms.

Packet classification can be seen as the categorization of incoming packets based on their headers according to specific criteria that examine specific fields within a packet header. The criteria are comprised of a set of rules that specify the content of specific packet header fields to result in a match. A packet classifier can be implemented in either software or hardware. An example of a real classifier in four dimension is presented in Table 2.1. In this table, the third column 'eq' and 'gt' keywords are operations that mean equal and greater than.

Rules	Destination IP (address mask)	Source IP (address mask)	Port No.	Protocol No.	Action
R1	192.168.190.69 255.255.255.255	192.168.80.11 255.255.255.0	*	*	Deny
R2	192.168.3.0 255.255.255.0	192.168.200.157 255.255.255.255	eq www	tcp	Deny
R3	192.168.198.4 255.255.255.255	192.168.160.0 255.255.255.0	gt 1023	udp	Permit
R4	0.0.0.0 0.0.0.0	0.0.0.0 0.0.0.0	*	*	Permit

Table 2.1: Sample classifier rules.

In this classifier, the first rule R1 has the highest priority and rule R4 has the lowest priority. An examples of packet classification is presented in Table 2.2.

Packet Header	**Destination Address**	**Source address**	**Port No.**	**Protocol No.**	**Action**
P1	192.168.190.69	192.168.80.11	www	tcp	Deny
P2	192.168.3.0	192.168.200.157	www	udp	Permit
P3	192.168.198.4	192.168.160.10	1025	tcp	Permit

Table 2.2: Example of packet classification.

The packet classification problem is inherently hard from a theoretical stand-

Rules	Destination IP (address mask)	Source IP (address mask)	Port No.	Protocol No.	Tuple space
R1	192.168.190.69 (255.255.255.255)	192.168.80.11 (255.255.255.0)	*	*	[32,32,0,0]
R2	192.168.3.0 (255.255.255.0)	192.168.200.157 (255.255.255.255)	eq www	tcp	[24,32,2,1]
R3	192.168.198.4 (255.255.255.255)	192.168.160.0 (255.255.255.0)	gt 1023	udp	[32,32,1,1]
R4	193.164.0.0 (255.255.0.0)	193.0.0.0 (255.0.0.0)	eq www	udp	[16,8,2,1]
R5	192.168.0.0 (255.255.0.0)	192.0.0.0 (255.0.0.0)	eq www	tcp	[16,8,2,1]
R6	0.0.0.0 (0.0.0.0)	0.0.0.0 (0.0.0.0)	*	*	[0,0,0,0]

Table 2.3: Simplified example of rule classification.

point [19]. It has been shown that the packet classification requires either $O(log\left(N^{(K-1)}\right))$ processing time and linear memory size, or $log(N)$ processing time and $O(N^K)$ memory size, where N is the number of rules, and K is the number of fields in header that used in rules [19], [67]. Most practical solution either use linear time to search through all rules sequentially, or use a linear amount of parallelism[19]. In general, there have been two major solutions for packet classification: hardware, and software. A few pioneering groups of researcher offered a collection of algorithmic solutions [20], [50], [51], [67], [96], [106]. There are some limitation to meet a good performance in high speed link in algorithmic solutions, therefore, architectural solution to the problem were proposed. These solutions are as follows: exhaustive search [51], [106], decision tree, grid-of-tries [96], [98] decomposition [51], and tuple space search.

Tuple space search

A high-level approach for multiple field search employs tuple spaces with a tuple representing information in each field specified by the rules. Srinivasan, et. al., introduced the tuple space approach and the collection of tuple search algorithms in [97], [98]. We provide a simplified example rule classification on five fields in Table 2.3. Address prefixes cover 32-bit addresses and port ranges cover 16-bit port numbers. For address prefix fields, the number of specified bits is simply the number of non-wildcard bits in the prefix. For the protocol fields, the value is simply a boolean: '1' if a protocol is specified, '0' if a wildcard is specified [98], [106]. The number of specified bits in a port range are less straightforward to define. The authors introduced the concept of nesting levels and Range-IDs to define the tuple value for port ranges. The nesting level specifies the layer of the hierarchy and the Range-ID uniquely labels the range within its layer. In this manner, all port ranges can be converted to a (Nesting level, Range-ID) pair. We present in the following an example to illustrate Range-IDs. The full range, in this example (0-65535) always has

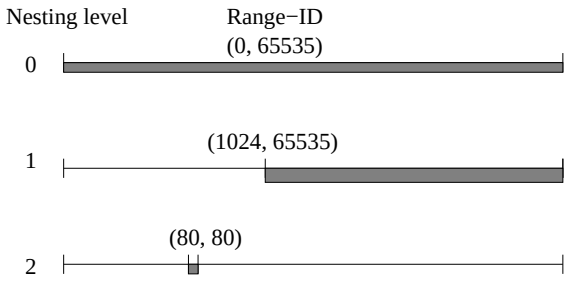


Figure 2.11: Assigning values for ranges, based on the Nesting Level and the Range-ID.

the id 0. The two ranges at level 1 namely, $(0, 1023)$ and $(1024, 65535)$ in our example receive id 0, and 1, respectively. The example of mapping a port range to a nesting level and a Range-ID for Table 2.3 is depicted in Figure 2.11.

In the following, we illustrate how a search key is constructed from a packet based on a tuple. A search key for the tuple [8, 24, 2, 0, 1] is constructed by concatenating the first octet of the packet source address, the first three octets of the packet destination address, the Range-ID of the source port, the range at nesting level 2 covering the packet source port number, the Range-ID of the destination port range at nesting level 0 covering the packet destination port number, and the protocol field. Finally, all algorithms using the tuple space approach involve a search of the tuple space or a subset of the tuples in the tuple space.

2.2.4 The Bloom Filter Concept

A Bloom filter is a simple space efficient randomized data structure to represent a set in order to support membership queries. It was introduced by Burton Bloom [25], [65], [95]. A Bloom filter is frequently utilized in network processing (areas), such as packet classification, packet inspection, forwarding, P2P networks, and distributed web caching [27], [30], [39]. The major variations of Bloom filters include the following: compressed Bloom filter [76], counting Bloom filter [43], distance-sensitive Bloom filter [60], Bloom filter with two hash functions, space-code Bloom filter [64], spectral Bloom filter [34] and dynamic Bloom filter [49] that are utilized in different network applications [49].

Standard Bloom filter

The elements of set S are $(x_1, x_2, ..., x_n)$ is represented by an array V comprising m bits that are initially all set to 0. A set of k independent hash functions h_1, h_2, ..., h_k (each with an output range between 1 and m) is utilized to set k bits in array V at positions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ for all x in set S. More precisely, for each element $x \in S$, the bits at positions $h_i(x)$ are set to 1 for $1 \leq i \leq k$. Moreover, a location can be set to 1 multiple times. To verify whether an item y is a member of the set S, the same set of hash functions is utilized to determine $h_i(y)$ (for $1 < i < k$) indicating the locations in array V to be checked whether their content is a 1. If one of these location yields a 0, y is certainly not a member of the set S. If all locations yield a 1, there is a high probability that y is a member of the set S (*positive*). However, as increasingly more bits in array V are set to 1, one can imagine that the probability of a false positive increases. It must be clear now that there is an inverse relation between the number of bits in the array and the false positive rate. In the extreme case, when all bits in the array are set to 1, every search will yield a (false) positive. The false positive probability is given as follows:

$$p_f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{2.6}$$

In this equation, n represents the number of elements, m represents the number of bits in the bit array and k represents the number of hashing functions. For a given m and n, the value k (the number of hashing functions) that minimizes the probability is as follows:

$$k = \frac{m}{n} ln2 \approx 0.7\frac{m}{n} \tag{2.7}$$

An example of a Bloom filter is depicted in Figure 2.12.

Figure 2.12 depicts the creation of a Bloom filter for a set of two items $R1$ and $R2$ and the subsequent testing whether $P1$ and $P2$ are part of the set. Each item R_i is hashed k times (using k independent hashing functions) and the corresponding bits are set to 1. To check whether $P1$ or $P2$ is member of the set, they are hashed with the same k hashing functions to determine the locations in the array to check whether these locations were set. For $P1$, it is clear that it is not part of the set.

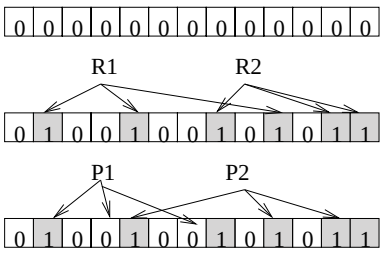


Figure 2.12: An example of Bloom filter array and its use.

Counting Bloom filter

The standard Bloom filter works fine when the members of the set do not change over time. When they do, adding items requires little effort since it only requires hashing the additional item and setting the corresponding bit locations in the array. On the other hand, removing an item conceptually requires unsetting the '1's in the array, but this could inadvertently lead to removing a '1' that was the result of hashing another item that is still member of the set. To overcome this problem, the counting Bloom filter was introduced [43]. In the counting Bloom filter, each bit in the array is replaced by a small counter. When inserting an item, each counter indexed by the corresponding hash value is incremented and, therefore, a counter in this filter essentially represents the number of items hashed "to it". When an item is deleted, the corresponding counters are decremented. In the following, we utilize $c(i)$ to denote the counter value associated with each i^{th} counter. Considering a Bloom filter for n items, with k hashing functions, and m counters, the probability that the i^{th} counter is incremented j times is given as a binomial random variable in the following:

$$p\left(c(i)=j\right)=\left(\begin{array}{c} nk \\ j \end{array}\right)\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j} \tag{2.8}$$

When using n-bit counters, an n-bit counter will overflow if and only if it reaches a value of 2^n. The analysis performed by Fan, et al., [43] shows that a 4-bit counter is adequate for most applications. The probability for 4-bit counter to overflow is:

$$p\left(max_i\langle c(i)\geq 16\rangle\right)\leq 1.37\times 10^{-15}m \tag{2.9}$$

2.3 Related Work

In this section, we take a brief look at previous works regarding the collaboration of reconfigurable processors in grid computing and network processing.

2.3.1 Collaboration of Reconfigurable Processors in Grid Computing

In [77], the design and implementation of a metacomputer based on reconfigurable hardware was presented. The Distributed Reconfigurable Metacomputing (DRMC) is defined as "the use of powerful computing resources transparently available to the user via a networked environment". The DRMC provides an environment in which computations can be constructed in a high-level manner and executed on clusters containing reconfigurable hardware. In the DRMC architecture, applications are executed on clusters using the condensed graphs model of computation that allows the parallelism inherent in applications to be executed by representing them as set of graphs.

In [31], a limited flooding approach for mobile ad hoc networks was proposed that utilized neighbors information to limit broadcast redundancies. It reduced control overhead of ad hoc routing protocols and introduced some benefits including efficient flooding, density, and mobility adaptation. A set of rules and guidelines for the implementation of Distributed Processing Network (DPN) as the basis for a dynamic reconfigurable architecture have been presented in [111]. The DPN presents the general architecture to solve complex problems using reconfigurable computing.

Performance of reconfigurable architectures for image processing applications was presented in [23]. The impact of hardware capacity, reconfiguration time, memory organization, and bus bandwidth of FPGA-based systems on the performance were studied. A speedup of up to 87 has been achieved over a GPP. Performance of reconfigurable elements for gray level co-occurrence matrix (GLCM) and Haralick texture feature for image sizes 512×512, 1024×1024 and 2048×2048 was presented in [103]. The speedups of 4.75 and 7.3 were obtained when compared with a general-purpose processor for GCLM and Haralick co-occurrence matrix, respectively. The target hardware for this work was Celoxica RC1000-PP PCI based FPGA development board equipped with a Xilinx XCV2000E Virtex FPGA. In addition, a co-occurrence matrix media kernel has been implemented on the various FPGA devices such as Virtex2 and Spartan3 and on a media-enhanced GPPs using MMX technology in [55].

Minimum speedups of 20 were obtained using FPGA implementations over media-enhanced GPPs, while the maximum speedups exceeds 100.

A performance model for fork-join class and Synchronous Iterative Algorithm (SIA) was presented in [93]. They considered division of computation between the workstation processor and the reconfigurable unit. They focused on algorithms and applications that fit into the fork-join class and SIAs types. The 2D-FFT application has been implemented on both the standard cluster and the prototype Adaptable Computing Cluster (ACC) in [110]. The ACC is an architecture that attempts to improve high-performance cluster computing with FPGAs, but not by merely adding reconfigurable computing resources to each node. Rather, by merging cluster and reconfigurable technologies and enhancing the commodity network interface. A system infrastructure that allows local mobile devices to interact with the grid is introduced in [48]. A proxy with the ability of dual communication to transfer the request from the mobile device to the grid is the main part of that architecture.

2.3.2 Network Packet Processing

Several research groups proposed a collection of software and hardware solutions in packet classification area[19], [51], [67], [106]. These solutions are: exhaustive search, decision tree, grid-of-tries, decomposition and tuple space search [51], [98], [106]. Many algorithms exist in the packet classification area and we discuss here only those algorithms that are related to our work. In [96], [97], [98], were presented the tuple space approach and the collection of tuple search algorithms. A high level approach for multiple field search employs tuple space. A tuple defines the number of specified bits in each field of the rule. The tuple-based algorithms utilize traditional hashing system. In [30], a cache design based on the standard Bloom filter is investigated and has been extended to support aging (adding the ability to evict stale entries from the cache), bound misclassification rates, and use multiple binary predicates. It examines the exact relationship between the size and dimension of the number of flows that can be supported and the misclassification probability incurred. Additionally, it presents extensions for gracefully aging the cache overtime to minimize misclassification. In [39], [95], an extended version of the Bloom filter is introduced. It presents a novel hash table architecture and lookup algorithm and converts a Bloom filter into a counting Bloom filter and associated hash bucket which improves the performance over a standard hash table by reducing the number of memory accesses needed for the most time-consuming lookups. It needs to reconsider all the items for each inserted item that consequently leads to longer

processing time. In [40], [39], an approach to packet classification which combines architectural and algorithmic techniques is presented. The starting point is crossproduct algorithm which is fast but has significant memory overhead due to the extra rules needed to represent the crossproducts. It modifies the crossproduct method to reduce the memory requirement. Unnecessary accesses to the off-chip memory are avoid by filtering them through on-chip Bloom filters.

Compressed Bloom filters were introduced in [76], which improved performance when the Bloom filter is passed as a message in a distributed protocols. The author investigated compressed Bloom filter for distributed proxy web caching and showed that by using compressed Bloom filters, proxies can reduce the number of bits broadcast, the false positive rate, and the amount of computation per lookup.

In [117], a hash architecture called a Multi-predicate Bloom-filtered Hash Table (MBHT) using parallel Bloom filters is presented. It is generated off-chip memory addresses in the base-2^x number system, $x \in \{1, 2, ...\}$, which removes the overhead of pointers. Using a larger base of number system, an MBHT reduces on-chip memory size. A SPSwitch as a novel switching engine to make wire speed forwarding decisions on flat information labels introduced in [42]. It has been addressed some part of scalability issues in a data-oriented forwarding layer for state reduction and line speed operations. SPSwitch has been designed based on the Bloom filter systems. In [41], introduced the retouched Bloom filter (RBF). The main idea is to remove each false positive by resetting a carefully chosen bit in the bit vector that makes up the Bloom filter. It analytically showed that the trade-off between false positives and false negatives is at worst neutral, on average, when randomly resetting bits in the bit vector, whether these bits correspond to false positives or not. In [57], [58], [59], a two-stage pipelined Bloom filter was proposed for network intrusion detection. It is shown that the smaller number of hashing functions implemented in the first stage of a pipelined Bloom filter, the more the power saving is. The authors also examined three type of hashing functions to observe the effect of power saving. The challenge in the two-stage pipelined Bloom filter is related to determine number of hashing functions in each stage. For each incoming key the configuration (number of hashing functions) in each stage should be changed or number of hashing function can be fixed. In the first case, some configuration overheads is needed and in the second case the configuration does not operate efficiently. In [86], the power, latency, and area characteristics for two counting Bloom filters implementations using full custom layouts in a commercial $0.13\ \mu m$ technology were investigated. Their first implementation

is based on a SRAM array of counts and the second is based on an array of linear feedback shift register counters.

In [109], ASAN (Active System Area Network) is presented as a project dealing with the combining the networking with computing together. They have studied the ways to build a much more powerful network interface which is capable of doing a lot of networking related computing task. The main idea is to utilize the FPGA to construct active SAN that can perform the computation together with communications when data is in transit. They have got good results by making experiments on a Myrinet base system. The Active SAN focuses in the SAN architecture to enhance the performance of cluster computer only. In [116], the architecture for QoS-enable router and grid-oriented supercomputer (AQRGS) focuses on constructing a new architecture for emerging grid applications and at the same time, improving the performance of the grid node supercomputer. It utilizes the network processor technology to offload the communication overhead from the CPUs and improve the computing performance of the router.

In [82], the StepNP is introduced an exploratory network processor simulation environment for exploring applications, multiprocessor network processing architecture, and system on chip (SOC) tools. The StepNP is modeled at the functional and transaction level and not at cycle accurate level. In [87], a queuing model analysis is presented as a valuable tool for investigation the performance and operating characteristic of communication networks and computer systems. In [71], the optimal capacity allocation is considered in a clustered web system environment. It formulates the problem as a nonlinear program to minimize a convex separable function of the capacity assignment vector. The solution can be applied in e-commerce service environment that involves multiple clusters of machines and each cluster handles a particular set of functions. An approximation method to solve the problem was developed. In [83], the assignment of the service capacity in a queuing network is considered. The author studies systems with several types of incoming items, general service time distributions, stochastic or deterministic routing, and a variety of service regimes. The residual-life approximation technique for the distribution of queuing times was utilized. In [68], [108], analytical modeling using mean value analysis (MVA) has been used in shared memory multi-processor systems. This technique is shown to be efficient and reasonably accurate for large systems. It used the closed queuing model and an MVA analysis. In [72], J. Lu, et. al., proposed a performance analysis of network processor-based application design using the closed queuing model and an MVA algorithm.

2.4 Summary

In this chapter, some high-performance processing systems in the networked and grid environments were introduced. These high-performance processing systems focused on computing and communication. In the high-performance computing systems, the capabilities of grid computing and reconfigurable architectures were presented. Reconfigurable architectures combined the flexibility of a general-purposed system with the high-performance of application specific systems. We also explained the grid computing concept and its capabilities. The most important capabilities of grid computing are: exploiting under-utilized resources, parallel CPU capacity, and reliability.

In the communication, we focused on network processors, and Bloom filters to achieve higher performance in the network processing application such as packet classification. Furthermore, we presented the queuing modeling to evaluate the performance of network processors.

Chapter 3

Collaborative Reconfigurable Processors on Grid Computing

In the previous chapter, we introduced and described in details the advantages of reconfigurable processors and grid nodes as a computing resources in a high-performance processing system. Reconfigurable computing has proven to be a promising technology to increase the performance of certain algorithms in scientific and engineering applications in recent years. At the same time, in grid computing, a large pool of heterogeneous computing resources is geographically dispersed over a large network such as the Internet. A technique to achieve high-performance with flexibility is to utilize collaboration of reconfigurable processors in grid computing.

In this chapter, we propose Collaboration of Reconfigurable processors in Grid Computing (CRGC) (in Section 3.1). We analyze a lower and upper bounds of performance for this CRGC (in Section 3.2). In Section 3.3, we present the mapping of several computationally intensive multimedia kernels such as the 2D Discrete Wavelet Transform (DWT) and the co-occurrence matrix using the proposed approach. To investigate our idea, we extend a version of grid simulator (GridSim v4) that we termed the Collaborative Reconfigurable Grid Simulator (CRGridSim) to support reconfigurable processor modeling and the neighborhood concept on grid environment (in Section 3.4). We summarize this chapter in Section 3.5.

3.1 The Concept of Collaborative Reconfigurable Processors on Grid

In grid computing, a large pool of heterogeneous computing resources is geographically dispersed over a large network, e.g., the Internet. Our approach to achieve high-performance and flexibility is to utilize reconfigurable processors in grid computing. We termed the utilization of reconfigurable processors that collaborate together in a grid environment Collaborative Reconfigurable Grid Computing (CRGC). The general view of CRGC is depicted in Figure 3.1. This figure shows that the processing elements are the part of resources on grid computing. Each processing element can be either GPP or Reconfigurable Element (RE) also known as reconfigurable processor. The processing capability of GPP processing element in the simulation environment is defined in the form of Million Instructions Per Second (MIPS) for Standard Performance Evaluation Corporation (SPEC) benchmark. Moreover, the RE is defined by its specifications such as reconfiguration time, reconfiguration file size, and its speedup over a GPP. It should be noted that the RE is implemented as FPGA to execute an amenable application with inherent level of parallelism. The same application can take much longer time when executed on a GPP.

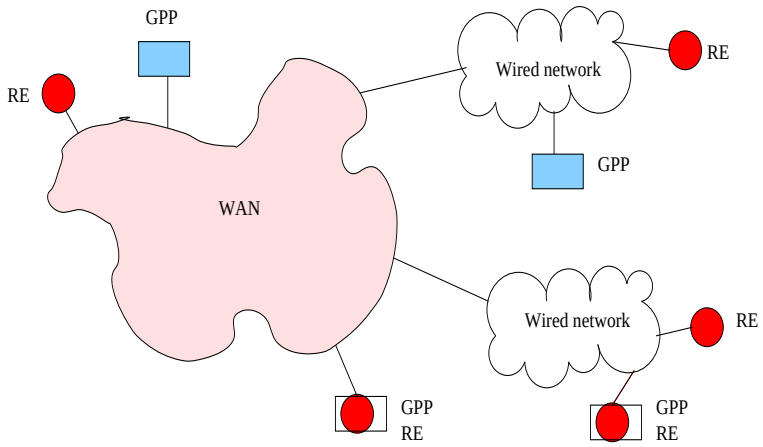


Figure 3.1: A general view of a collaborative grid environment with reconfigurable processors and general-purpose processors (GPPs). The lightblue box shows GPP and box with the circle inside shows GPP augmented with RE (reconfigurable element).

Processing elements offload part of their computational workload to reconfig-

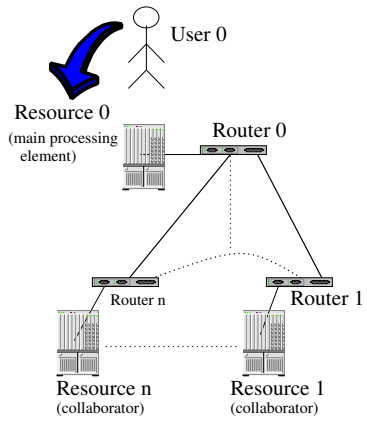
urable computing resources. In this type of computing, various software codes targeting different processing architectures are stored either in centralized or decentralized manner and must be distributed to the computing resources when needed.

In traditional grid computing, the grid is responsible for sending a task to other processing elements. The task submission is performed in the following ways [33]:

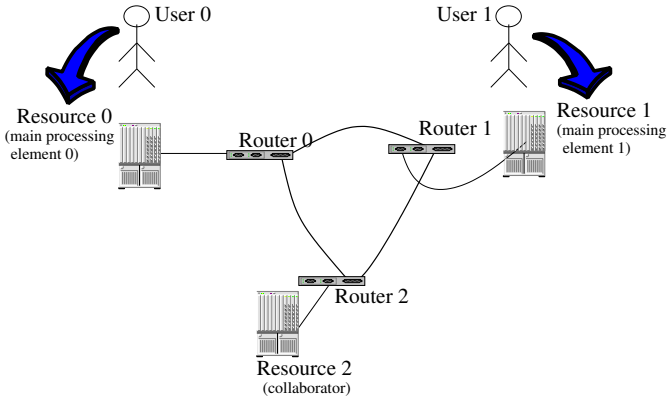
- Scheduling: The grid includes a task scheduler that automatically finds the most appropriate processing elements on which to run any given task that is waiting to be executed. Schedulers react to the current availability of resources on the grid.
- Reservation: The grid reserves the processing elements in advance to improve the quality of service.
- Scavenging: In a scavenging grid, each processing element that becomes idle would typically report its idle status to the grid management element.

On the other hand, in CRGC, processing elements communicate and collaborate together based on the *neighborhood concept* [8], [15], [115]. Each grid processing element requests assistance from neighboring processing elements. The tasks can be inserted into the grid through existing grid elements. In our implementation of the neighborhood concept, the neighbor processing elements are direct neighbor to a requesting grid element. The direct neighbor is defined as a grid element that is physically (or geographically closely) located next to the current requesting grid element. The network backbone can be seen as a collection of primitives. A primitive is defined as a set of collaborator processing elements, requesting processing elements with related communication links and its network equipments, e.g., routers and switches. Two important primitives are depicted in Figure 3.2.

Figure 3.2 (a) depicts a primitive with one requesting processing element and n collaborating processing elements. In this figure, resource 0 is requesting processing element and resource 1 to resource n are collaborator processing elements. The primitive in Figure 3.2 (a) includes (resource 0, router 0, resource 1, router 1, resource n, and router n). A primitive with two requesting processing elements and one collaborating processing element is depicted in Figure 3.2 (b). The neighborhood concept with active primitives in the real grid is depicted in Figure 3.3.



(a)



(b)

Figure 3.2: Basic primitives that are utilized in the neighborhood concept. (a) A primitive with one requesting processing element and n collaborator processing elements. (b) A primitive with two requesting processing elements and one collaborator processing element.

Based on Figure 3.3, we can observe that each user and the related requesting processing element can find the correspondent neighbor processing element. For example, user 0 and resource 0 can operate based on primitive in Figure 3.2 (a). From Figure 3.2, in the first scenario, resource 0 is assisted by resource 1 and in the second scenario, resource 0 is assisted by resources 1 and 2. We have a similar condition for user 1, in this case resource 3 gets help from resource 5.

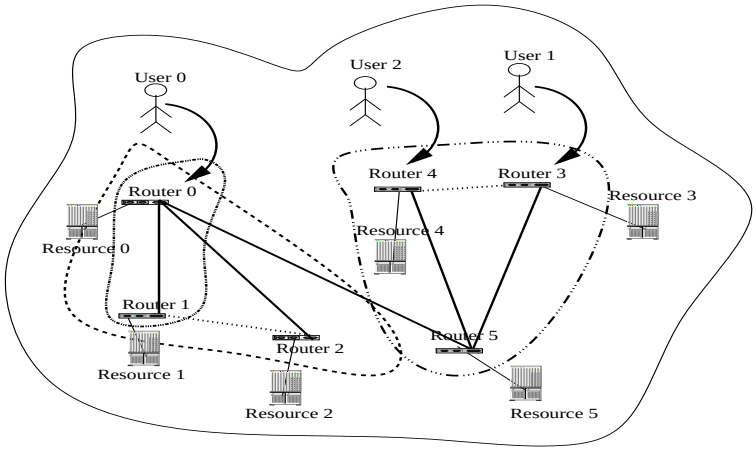


Figure 3.3: Active primitives in the sample network.

3.2 Performance Model Analysis

This analysis is performed to determine the performance bounds in term of execution time for the collaborative reconfigurable processors on grid environments. The maximum and minimum execution times show the lower and upper bounds of performance, respectively. Table 3.1 describes all notations and symbols which have been used in these equations.

The total processing time in a non-collaborative system without inter-task dependency can be represented as Equation (3.1).

$$t_{non-col} = \sum_{i=1}^{n} M_i t_{GPP} \tag{3.1}$$

where $t_{non-col}$, M_i, and t_{GPP} are the processing time in non-collaboration system, the number of instructions of subtask i (each submitted task is broken to several subtasks), and the processing time of each instruction by a GPP, respectively. In addition, the total processing time in a collaborative system is defined as Equation (3.2). This is because, in the CRGC system some of tasks is processed by GPP and others are distributed among collaborator processing elements. Therefore, the total processing time is the summation of processing time by GPP and maximum of processing time by the collaborator processing elements.

Symbol	Definition
Bw	Network bandwidth
k	Speedup factor of a reconfigurable processor over a GPP
k_j	Speedup factor of reconfigurable processor j over a GPP
k_{max}	Maximum speedup factor of a reconfigurable processor over a GPP
M_i	Number of instructions for subtask i
n	Number of subtasks (gridlets)
$n1$	Number of subtasks that can be processed by neighbor collaborators
$n1_j$	Number of processed subtasks by collaborator j
n_{pkt}	Number of packets
p	Number of collaborator processing elements
rs	Reconfiguration speed
rfs	Reconfiguration file size
S	Size of all subtasks (Bytes)
s_i	Size of subtask i (Bytes)
s_{Pkt}	Packet size
t_{col}	Total execution time in a collaboration system
t_{com}	Communication time between different processing elements
t_{GPP}	Processing time for each instruction by a GPP
t_{max}	Maximum execution time of an application on CRGC
t_{min}	Minimum execution time of an application on CRGC
$t_{non-col}$	Total execution time in a non-collaboration system
t_{PE_j}	Processing time of a subtask using collaborator processing element j
t_{ptt}	Transmission time of a packet
t_{RE}	Processing time for each instruction by a RE that is equal to t_{GPP}/k
t_{rec}	Reconfiguration time

Table 3.1: Performance analysis symbols and their definitions.

$$t_{col} = \sum_{i=1}^{n-n1} M_i t_{GPP} + max\left\{t_{PE_j},\ j = 1..p\right\} \ \ and\ \ n1 = \sum_{j=1}^{p} n1_j \quad (3.2)$$

where t_{col} is the processing time in a collaborative system, $n1$ is number of subtask that is processed by the collaborator processing elements, and $n1_j$ is a set of processed subtasks by an individual collaborator element. Moreover, t_{PE_j} represents the processing time using collaborator processing element j that is defined using the following Equation (3.3).

$$t_{PE_j} = \left(\sum_{i=start_j}^{end_j} M_i t_{RE_j}\right) + t_{rec_j} + t_{com_j} = \frac{\sum_{i=start_j}^{end_j} M_i}{k_j} t_{GPP} + \frac{rfs_j}{rs_j} + t_{com_j} \quad (3.3)$$

In this equation, the subtasks from $start_j$ to end_j ($end_j - start_j = n1_j$) are processed by collaborator j. The size of the reconfiguration file and the reconfiguration speed are represented by *rfs* and *rs*, respectively. In addition, the term t_{rec_j} is reconfiguration time needed to reconfigure the collaborator processing element j. If the submitted tasks cannot distribute on the collaborator elements then $t_{com} = 0$, $t_{rec} = 0$, $t_{col} = 0$, and $n_1 = 0$ in Equation (3.3). As a result, $t_{PE_j} = 0$ and Equation (3.2) will be the same as the Equation (3.1). Therefore, the main processing element (GPP) executes all submitted subtasks without any assistance from neighbor collaborator processing elements. Consequently, the lower bound of performance in term of maximum execution time (t_{max}) is represented as:

$$t_{max} = \sum_{i=1}^{n} M_i t_{GPP} = t_{non-col} \tag{3.4}$$

To evaluate the upper bound, Equation (3.2) should be minimized. In this case, some submitted tasks are executed by reconfigurable processors and the rest are executed on main processing element. The upper bound of performance in term of minimum execution time is represented as follows:

$$t_{min} = min\left\{\sum_{i=1}^{n-n_1} M_i t_{GPP} + max\left\{t_{PE_j},\ j = 1..p\right\}\right\} \tag{3.5}$$

The minimization of this equation depends on different parameters such as number of collaborators elements, length and number of submitted subtask, the reconfiguration time, the bandwidth of the network, the propagation delay, and scheduling algorithm. To simplify the math analysis, we utilize the transitivity property of inequalities to achieve the upper bound of performance [45].

$$\begin{aligned} &t_{min} > min\{t_{PE_j}\} = min\left\{\sum_{i=start_j}^{end_j} M_i t_{RE_j} + t_{rec_j} + t_{com_j}\right\} = \\ &min\left\{\frac{\sum_{i=start_j}^{end_j} M_i}{k_j} t_{GPP} + \frac{rfs_j}{rs_j} + t_{com_j}\right\}\ \ and\ \ end_j - start_j = n1_j\ and\ j = 1..p \end{aligned} \tag{3.6}$$

We make some reasonable assumptions, for e.g., t_{rec} and t_{com} are assumed to be very small and these parameters as said before can be ignored. Consequently, Equation (3.6) is represented as:

$$t_{min} > min\left\{\sum_{i=start_j}^{end_j} M_i \frac{t_{GPP}}{k_j},\ \ j = 1..p\right\} \tag{3.7}$$

The right side of Equation (3.7) includes $end_j - start_j = n1_j$ subtasks while for p collaborator processing elements with n subtasks this equation is rewritten as follows.

$$min\left\{\sum_{i=start_j}^{end_j} M_i \frac{t_{GPP}}{k_j},\ \ j = 1..p\right\} \approx \frac{1}{p} min\left\{\sum_{i=1}^{n} M_i \frac{t_{GPP}}{k_j},\ \ j = 1..p\right\} \tag{3.8}$$

The equation above is minimized when the denominator of the second term of the inequality has the maximum value. The maximum value is achieved when a reconfigurable processor with the highest speedup is employed. We can rewrite the Equation (3.7) as follows.

$$t_{min} > \frac{1}{p}\left\{\sum_{i=1}^{n} M_i \frac{t_{GPP}}{max\{k_j\}}\right\} = \frac{1}{k_{max}p}\left\{\sum_{i=1}^{n} M_i t_{GPP}\right\} = \frac{1}{k_{max}p} t_{non-col} \tag{3.9}$$

This equation shows the lower bound for the execution time in a collaborative system is k_{max} times less than the execution time in a non-collaborative system, where k_{max} is maximum speedup of a reconfigurable processor over a GPP. Using Equation (3.4) and Equation (3.9) the lower and upper bounds of the performance is expressed using the following equation.

$$\frac{1}{k_{max}p} t_{non-col} < t_{col} \leq t_{non-col} \tag{3.10}$$

3.3 Multimedia Kernels

In this section, we briefly describe a motivation why we selected multimedia kernels, and subsequently the two chosen multimedia kernels: the 2D DWT and the co-occurrence matrix.

3.3.1 Motivation on Architectures for Multimedia Kernels

In this section the main motivation to propose collaboration of reconfigurable processors in grid computing for multimedia kernels is explained.

Many architectures ranging from application-specific processors to domain-specific processors have been proposed to process multimedia applications [89]. The Application-Specific Integrated Circuits (ASIC) approaches offer the advantages of high-performance, but their design and debugging phases involve a significant amount of time. Because the development cost cannot be spread across multiple applications. In addition, they are suitable only for specific functions, and future extensions are not possible without redesigning the hardware. The GPPs enhanced with Single-Instruction Multiple-Data (SIMD) extensions provide programmability compared to dedicated architectures, while GPPs equipped with SIMD extensions need many overhead instructions such as *packing/unpacking* and data *re-shuffling* instructions [90].

Advanced dedicated multimedia processors use Very Long Instruction Word (VLIW) architectural schemes to exploit a high degree of Instruction-Level Parallelism (ILP) [62]. This is because VLIW architectures have many advantages compared to superscalar processors. For example, VLIW processors employ static instruction scheduling performed at compile-time rather than dynamic scheduling performed at run-time as in superscalar processors, which requires much more hardware [44]. Furthermore, hardware does not need to determine which instructions can be issued in parallel.

Another related dedicated architecture to process multimedia applications is the Imagine processor [80], which has a load/store architecture for 1D streams of data records. Imagine is a stand-alone multimedia coprocessor. The focus of the Imagine project is to develop a programmable architecture for graphics and image/signal processing.

Recently, heterogeneous multicore Cell processor has been developed by a partnership of IBM, Sony, and Toshiba for computational intensive applications such as multimedia [56]. Cell is a heterogeneous chip multiprocessor consisting of a PowerPC core that controls eight high-performance Synergistic Processing Elements (SPEs). Each SPE has one SIMD computation unit that is referred to as Synergistic Processor Unit (SPU). In order to bring multimedia data in form amenable for SIMD processing in Cell processor, many overhead instructions are needed. In addition, available compilers cannot exploit efficiently SIMD vectorization automatically [89].

None of the proposed architectures, however, can provide high-performance

with flexibility for multimedia applications. In other words, the requirements of multimedia applications have been not matched well with the ability of the existing architectures. This is because the nature of multimedia applications is dynamic. They use a variety of multimedia algorithms, process different media data such as text, handwritten data, image, video, and 3D data in different processing environments ranging from desktop systems to mobile systems. In addition, multimedia applications are very computationally intensive. For example, some multimedia applications involve execution of complex algorithms such as 3D video rendering, scalable video coding, and stereo vision in real-time.

Therefore, we propose a parallel architecture to process multimedia applications using collaboration of reconfigurable processors in grid computing. Collaboration of different architectures should be used in order to meet the computational demand of multimedia processing. Grid computing can provide a solution to address issues related to handling large volume data and executing computational intensive multimedia kernels efficiently. Subsequently, we present two multimedia kernels the 2D DWT and co-occurrence matrix. We selected these multimedia kernels because they are compute-intensive. For example, the results [91] show that the 2D DWT consumes on average 46% of the encoding time for lossless compression. For lossy compression, the 2D DWT on average even requires 68% of the total encoding time.

3.3.2 Discrete Wavelet Transform

The digital wavelet representation of a discrete signal X consisting of N samples can be calculated by convolving X with the lowpass and highpass filters and down-sampling the output results by 2, so that the two frequency bands each contains $N/2$ samples. With the correct choice of filters, this operation is reversible. This process decomposes the original image into two sub-bands: the lower and the higher bands [100]. This transform can be extended to multiple dimensions by using separable filters. A 2D DWT can be performed by first performing a 1D DWT on each row (*horizontal filtering*) of the image followed by a 1D DWT on each column (*vertical filtering*). The digital wavelet transform is mainly used for image and video compression. Standards such as JPEG2000 [84] are based on the 2D DWT. For our investigation we selected the Daubechies' transform with four coefficients [107]. Figure 3.4 depicts the C implementation of the Daub-4 transform.

Specification of the input data: An image with optional size, the data type of each element is float. Computation: ALU operation: addition, Float operation:

```
void DWT_Daub_4() {
  int i, j, jj, ii;
  float low[] ={-0.1294, 0.2241, 0.8365 , 0.4830};
  float high[]={-0.4830, 0.8365, -0.2241, -0.1294};

  for (i=0; i<N; i++)
    for(j=0, jj=0; jj<M; j++, jj +=2) {
      tmp[i][j]  = img[i][jj]     * low[0] +
                   img[i][jj + 1] * low[1] +
                   img[i][jj + 2] * low[2] +
                   img[i][jj + 3] * low[3];

      tmp[i][j + M/2] = img[i][jj]     * high[0] +
                        img[i][jj + 1] * high[1] +
                        img[i][jj + 2] * high[2] +
                        img[i][jj + 3] * high[3];
  }
  for (i=0, ii=0; ii<N; i++, ii +=2)
    for (j=0; j<M; j++) {
      img[i][j]= tmp[ii][j]   *low[0]+tmp[ii+1][j]*low[1] +
                 tmp[ii+2][j] *low[2]+tmp[ii+3][j]*low[3];

      img[i+N/2][j]= tmp[ii][j]  *high[0]+tmp[ii+1][j]*high[1] +
                     tmp[ii+2][j]*high[2]+tmp[ii+3][j]*high[3];
  }
}
```

Figure 3.4: The 2D DWT using the Daub-4 for an image of size $N \times M$.

addition and multiplications Output is an image so that each element is float.

3.3.3 Co-Occurrence Matrix

Texture features are usually used in image and video processing. Texture features determine the dependencies between neighboring pixels within a region of interest in an image [37]. Haralick et al. [52] defined some texture features which utilize co-occurrence matrices. The features are related to second order statistics of neighboring pixels at different directions and distance. The number of occurrences of two neighboring pixels with a distance d and with a certain direction is stored in a co-occurrence matrix. The co-occurrence matrix is always a square matrix of size $N_{gl} \times N_{gl}$, where N_{gl} is the number of available gray levels in the image. Consequently, the size of this matrix is dependent from distance and direction neighboring pixels and also from the

image size. Co-occurrence matrix consists of relative frequencies $P(i, j; d, \delta)$ of two neighboring pixels i, j separated by distance d at orientation δ in an image.

In several image processing applications, for example medical images Haralick texture features are computed for some Region Of Interest (ROI). The size of the ROIs is not fixed and it is depended on the input image. As many ROIs are selected, therefore for each ROI a co-occurrence matrix is computed. There are eight directions, while only four of them are unique. the co-occurrence matrix will be symmetry if all directions are considered. Figure 3.5 depicts the C implementation of the co-occurrence matrix with considering eight different directions.

```
for(i=1; i < img_height-1; i++)
for (j=1; j < img_width-1; j++)
{
GLCM[ img[i][j] ][ img[i-1][j-1] ]++;
GLCM[ img[i][j] ][ img[i-1][j] ]++;
GLCM[ img[i][j] ][ img[i-1][j+1] ]++;
GLCM[ img[i][j] ][ img[i][j-1] ]++;
GLCM[ img[i][j] ][ img[i][j+1] ]++;
GLCM[ img[i][j] ][ img[i+1][j-1] ]++;
GLCM[ img[i][j] ][ img[i+1][j] ]++;
GLCM[ img[i][j] ][ img[i+1][j+1] ]++;
}
```

Figure 3.5: C implementation of the co-occurrence matrix.

3.4 Simulation Environment and Tools

In this section, we present our simulation environment and its configuration to simulate multimedia kernels. We investigate multimedia kernels as case study on a network where this network includes primitives to construct a backbone of reconfigurable processors on grid. In these primitives, different processing elements collaborate together to execute an application in a grid environment. Each processing element can be a GPP or a Reconfigurable Element (RE) (reconfigurable processor). The specifications of processing elements is defined in the form of Million Instructions Per Second (MIPS) for Standard Performance Evaluation Corporation (SPEC) benchmark. In here, we used 30, 35, 40 and 50 MIPS for processing elements.

The simulation environment is an extended version of GridSim (a traditional

Java-based discrete-event grid simulator) [28], [101]. The features of the GridSim toolkit include the following:

- It allows modeling of heterogeneous types of resources.
- Resources can be modeled operating under space- or time-shared mode.
- Resource capability can be defined (in the form of MIPS as per SPEC benchmark).
- Resources can be located in any time zone.
- Weekends and holidays can be mapped depending on resources local time to model non-Grid (local) workload.
- Resources can be booked for advance reservation.
- Applications with different parallel application models can be simulated.
- There is no limit on the number of application tasks that can be submitted to a resource.
- Application tasks can be heterogeneous and they can be CPU or I/O intensive.
- Multiple user entities can submit tasks for execution simultaneously in the same resource, which may be time -shared or space-shared. This feature helps in building schedulers that can use different market-driven economic models for selecting services competitively.
- Network speed between resources can be specified.
- It supports simulation of both static and dynamic schedulers.
- Statistics of all or selected operations can be recorded and they can be analyzed using GridSim statistics analysis methods.

We configured and prepared the GridSim simulator based on the multimedia kernels properties to support the collaborative processing between reconfigurable processors. This extension of GridSim is called CRGridSim. The main feature of CRGridsim is to simulate reconfigurable hardware. In CRGridsim, each application can be broken down into different subtasks called gridlets (subtasks). Each application is packaged as gridlets whose contents include the task length in Millions of Instructions (MIs). The task length is expressed in

terms of the items of the time it takes to run on a standard GPP [8]. To simplify the simulation of the proposed approach the following assumptions have been made. First, reconfigurable processors do not support partial reconfiguration. Second, there is not any background traffic on the network.

In order to understand how many instructions are required to execute the discussed multimedia kernels, we executed both the 2D DWT and co-occurrence matrix kernels using the SimpleScalar toolset [18] for an image of size 1024×1024. The number of committed instructions for the 2D DWT kernel is almost 46 MIs, while the number of committed instructions for the second kernel is almost 83 MIs. This means that in order to provide each value for the first decomposition level of the 2D DWT, 44 instructions should be processed, while for the second kernel 80 instructions should be processed for each pixel.

Size of images and their correspondent gridlets (the required instructions to process each image) are depicted in Table 3.2. Four groups of different images with various sizes have been collected. Each image is sent in uncompressed form to the processing elements. The required instructions to process each image is packed up as a gridlet (subtask) and is submitted to the related processing elements either GPP or RE. Table 3.3 depicts the specification of processing elements in terms of MIPS.

Image			**# of instructions (MIs) for each gridlet**		
# of Group	**Size**	**# of images**	**2D DWT**	**Co-occ. matrix**	**2D DWT+ Co-occ. matrix**
1	768×1024	10	35	64	99
2	1024×1024	10	46	84	110
3	1200×1600	10	84	156	240
4	2134×2848	10	267	493	760

Table 3.2: Images and their correspondence gridlets (subtasks) specifications for different multimedia kernels. MIs means million of instructions.

Processing elements	**MIPS**
Main GPP	30
Collaborator 1 (GPP or RE)	35
Collaborator 2 (GPP or RE)	50
Collaborator 3 (GPP or RE)	40

Table 3.3: The specification of processing elements in terms of MIPS.

The multimedia kernels have been simulated on the CRGC with different configurations using topologies in Figure 3.2 (A). For our case, one main GPP works with either 2 or 3 collaborator processing elements. In other words, we assumed 3 and 4 processing elements collaborate as part of a grid environment. Reconfigurable processors (elements) and GPPs are used as the collaborator

Parameter	Value
Maximum packet size	32 and 64 KBytes
User-router bandwidth	100 Mb/sec
Router-router bandwidth	1000 Mb/sec
Number of images	40 (Table 3.2)
Number of users	1
Size of images	Based on Table 3.2
PE specification (MIPS)	Based on Table 3.3
Minimum speedup for RE	5 in compared to GPP
Reconfiguration file size	3 Mb
Reconfiguration speed	3 Mb/sec
Reconfiguration time	1 sec
Number of bits per pixel	24 bit

Table 3.4: Specifications of the simulated environment.

processing elements. The specifications of the simulated environment and primitives are depicted in Table 3.4. The minimum speedup of reconfigurable processors over GPPs for multimedia kernels has been set to 5. The reason for this has been mentioned in the following. As we discussed in Section 2.3, a co-occurrence matrix media kernel has been implemented on the various FPGA devices in [55], [103]. Minimum speedups of 5 were obtained using FPGA implementations over media-enhanced GPPs, while the maximum speedups exceeds 100.

It should be noticed that for simulated topology the Routing Information Protocol (RIP) is executed by the simulator. The arbitrary number of collaborators can be used that is depends on the application requirements and available processing elements. However, increasing the number of processing elements increases the overhead, for example the communication time. In order to map the multimedia kernels on reconfigurable processors, the possible reconfigurable parameters should be identified. For example, variable wavelet filter lengths and variable wavelet decomposition levels are two reconfigurable parameters for DWT. Different steps to execute an application on CRGC is depicted in Figure 3.6.

In the first three steps, network topology and application mapping policy are defined and parameters such as network bandwidth, packet size and number of gridlets (subtasks) in the simulator are configured. Steps 4, 5, and 6 show the execution of gridlets by collaborator processing elements. Available collaborator processing elements are selected to execute a set of gridlets, therefore, main processing element packetizes each gridlet and sends to related collabora-

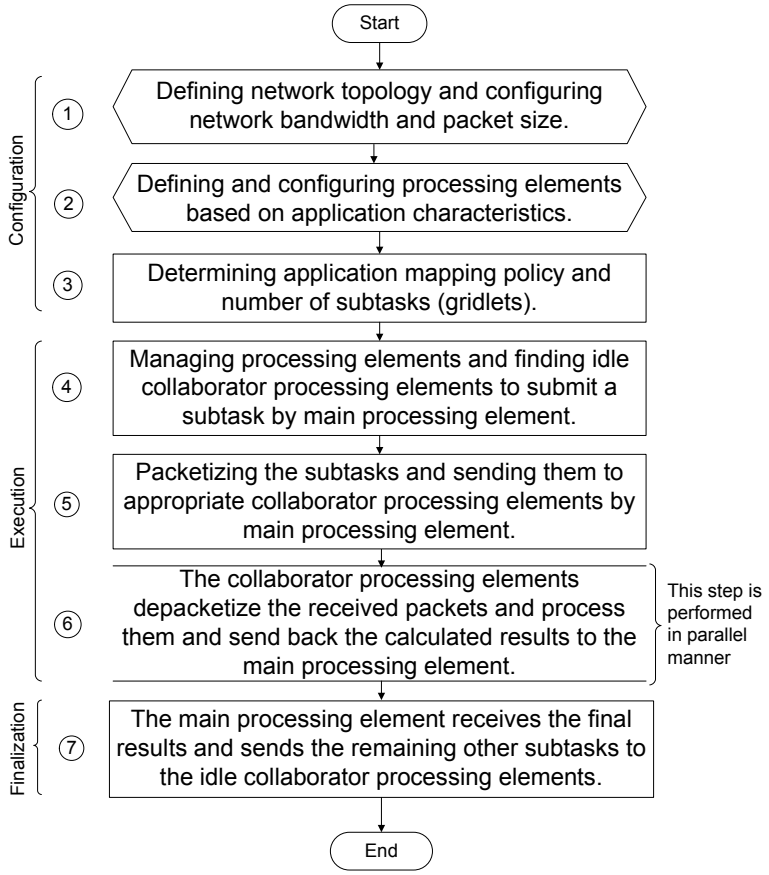


Figure 3.6: The flowchart of application execution in CRGridsim simulator.

tor processing elements. The target collaborator processing element collects received packets and executes the gridlets and finally send back results to main processing element. In step 7, main processing element receives calculated results and sends remaining gridlets to collaborator processing elements. The simulation results and performance evaluation are presented in Chapter 6.

3.5 Summary

This chapter presented the concept of collaborative reconfigurable processors in grid computing. More specifically, it is an approach to utilize reconfigurable computing resources in a grid environment to collaborate in achieving a single task or executing a single application. This approach intends to marry flexibility with high performance.

Reconfigurable computing provides much more flexibility than ASIC and much more performance than GPPs. Grid computing increases the performance of computationally intensive applications by exploiting the parallelism. Each part of an application can be executed on a processing element of a grid. The collaboration is implemented using neighborhood policy. In this policy, each processing element requests assistance only from its neighbor processing elements. In addition, we analyzed the lower and upper bounds of performance for the proposed architecture. We simulated the collaboration of four processing elements with different configurations on a grid computing. Furthermore, we mapped two computationally intensive multimedia kernels on the CRGC environments using CRGridsim simulator.

Chapter 4

Bloom Filter in High-performance Network Processing

In Chapter 2, we introduced the Bloom filter concept and its applications in network processing. Most network devices, e.g., routers and firewalls, require the processing of incoming packets (e.g., classification and forwarding) at wire speeds. These devices mostly incorporate special network processors that are comprised of a processor core with several memory interfaces and special co-processors that are optimized for packet processing. The gap between processor and memory performance has been a major source of concern for all of the computing society; this problem is exacerbated in packet processing systems. Such memory bottlenecks can be overcome by the following mechanisms: hiding of memory latencies through parallel processing and reducing the memory latencies by introducing a special memory architectures. One approach to achieve higher lookup performance is to utilize the Bloom filter data structure that is recently utilized in embedded memory technology in network processors.

In this chapter, we present three different approaches for Bloom filters. In Section 4.1, a cache architecture for counting Bloom filter is presented. A memory optimization approach for Bloom filter using an additional hashing function (BFAH) is introduced in Section 4.2. Section 4.3 presents a power efficient pipelined Bloom filter. Finally, Section 4.4, summarizes this chapter.

4.1 Cache Architecture for Counting Bloom Filter

A solution to decrease the processing time in the packet classification is the utilization of Bloom filters [12], [40], [66], [95]. There are numerous networking problems where such a data structure is required. Especially when space is an issue, a Bloom filter may be an excellent alternative to keep an explicit list.

In this section, we first describe the counting Bloom filter and analyze the probabilities of incrementing the counters. Afterwards, we introduce a new multi-level cache architecture called the *cached counting Bloom filter* (CCBF). In addition, the pruning procedure to optimize the memory utilization in the counting Bloom filter is described. Based on the counting Bloom filter analysis, we propose two multi-level cache architectures (an l-$level$ and a 3-$level$ one) and, subsequently, present the performance analysis. The performance metric is the number of accesses to different cache levels of the CCBF compared to the memory accesses when using the standard Bloom filter. In the 3-level cache, we further determine the size of cache levels for optimal false positive probabilities. To test the CCBF concept, we implemented a software packet classifier utilizing a 3-level CCBF employing tuple spaces that are traditionally utilized in hashing systems.

4.1.1 Pruned Counting Bloom Filter

In a counting Bloom filter, we compute k hash functions $h_1(), ..., h_k()$ over a set of items and increment the related k counters indexed by these resulting hash values. Subsequently, we store the item in the lists associated with each of the k buckets hence a single item is stored k times in memory. Therefore, it is needed to maintain up to k copies of each item requiring k times more memory compared to a standard hash table. However, in a Bloom filter only one copy need to be accessed while the other $(k-1)$ copies of item are never accessed, therefore, the memory requirement can be minimized in the mentioned architecture, resulting in the pruned counting Bloom filter. Figure 4.1 (a) depicts the results of hashing four items (rules). Additionally, in this figure, we introduce the concept of *buckets* that are pointed to by the counters storing the items (rules) of the set. The pruned counting Bloom filter for Figure 4.1 (a) is depicted in Figure 4.1 (b).

A method for pruning is to create a normal counting Bloom filter and only keeping items with a minimum value in their counter, and for the items with same counter, the item with lower index is selected. In this method, insertion

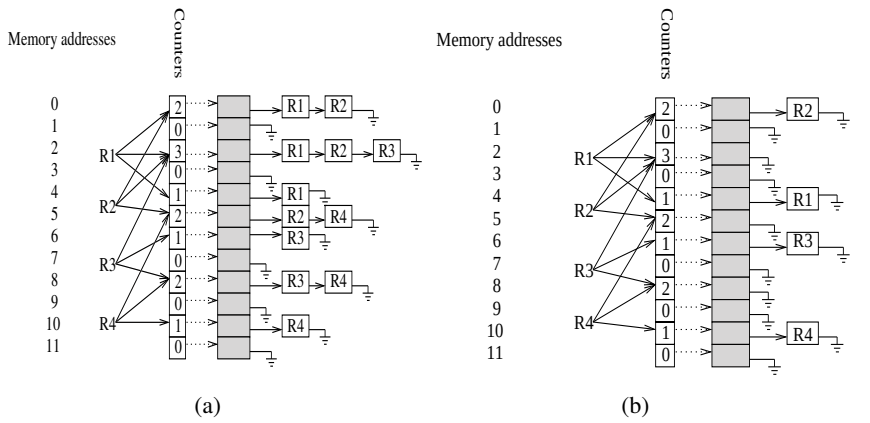


Figure 4.1: (a) The hash table architecture using counting Bloom filters for four items (rules). (b) The hash table using pruned counting Bloom filter.

and deletion of redundant items are preformed simultaneously. It must be noted that during the pruned counting Bloom filter creation, the counter values are not changed thereby, after pruning, the value of counters no longer expresses the number of items in the list and is greater than or equal to the number of items in each bucket (a memory buffer that are pointed to by the counters storing the items of the set). In the pruning procedure, all the other copies of an item except the one which is accessed during the search can be deleted. Therefore, after the pruning procedure we have one copy for each item and the result of this procedure is memory optimization. A limitation of the pruning procedure occurs when performing the sequential insertion since the value of counters after each insertion does not show the number of items in the bucket, and also changes the counters of the other items that were formerly hashed in to the buckets. This limitation is overcome by the searching the buckets that are pointed to by the hashing functions and then recalculating the addresses of items in these buckets. In other words, for inserting one item we must reconsider all items in those buckets [12], [95]. In addition, this pruning technique only works with counting Bloom filters which limits its application.

4.1.2 Cached Counting Bloom Filter Concept

According to definition of a Bloom filter, the number of hashing functions k with m counters and n items can be expressed as:

$$k = g\frac{m}{n} \tag{4.1}$$

where the value of g changes for different Bloom filter configurations. The optimal value for g to have a minimum false positive rate is $g = ln(2) \approx 0.6931$. From Equation (2.6) and Equation (4.1), the false positive rate for different value of g is depicted in Figure 4.2.

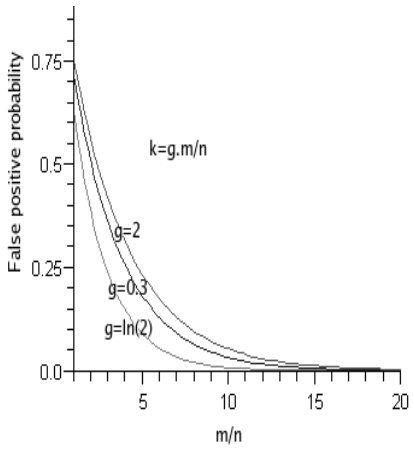


Figure 4.2: False positive probability for different configurations.

After the substitution of Equation (4.1) in Equation (2.6), we obtain the following equation:

$$p\left(c(i) = j\right) = \left(\begin{array}{c} gm \\ j \end{array} \right) \left(\frac{1}{m}\right)^{j} \left(1 - \frac{1}{m}\right)^{mg-j} \tag{4.2}$$

Using Equation (4.2), we can compute the probability of incrementing of the i^{th} counter for different values of g and m. Using Equation (4.2), the counter probability distribution for different counting Bloom filter configurations is depicted in Figure 4.3.

Based on the Figure 4.3, when $g \leq ln(2)$ the value of the counters with non-zero probability changes between 0 and 3, and when $g > ln(2)$ the value of

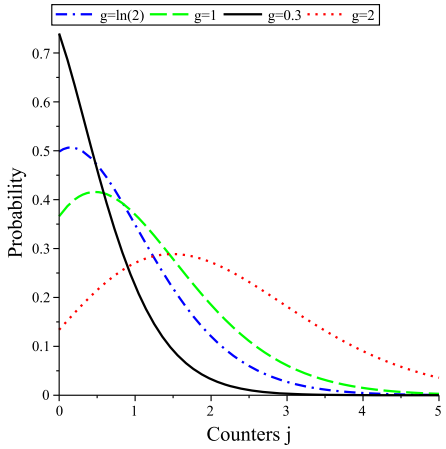


Figure 4.3: The counter probability distribution for different configurations in counting Bloom filters.

the counters with non-zero probability is increased (for $g = 2$, the value of the counters changes between 0 and 5). $g = ln(2)$ shows the optimal configuration with minimum false positive probability. In the other words, the most of the counters take the values between 0 and 3 when the counting Bloom filters configure using $g = len(2) \approx 0.6931$. Therefore, we can utilize a multi-level caching memory to store the items based on their counters. Subsequently, we introduce the cached counting Bloom filter (CCBF). The CCBF is slightly is different from real cache. It works based on the counting Bloom filter properties.

Cached counting Bloom filter analysis

In this section, we present the analysis of the cached counting Bloom filter (CCBF). The number of accesses to the memory depends on the fact whether the Bloom generates a 'positive' or 'negative' result. For the negative case, no accesses to the memory is needed since it is certain that they are not in the original set. For the positive case, still it must be verified whether the item in question is a member or not (false positive). Consequently, we assume in the analysis that all tests are on different elements which would result in the testing of n elements (the same number of items in the original set). The number of accesses in a standard Bloom filter is $nk(1 + p_f)$ memory accesses, where n represents the number of items, k represents the number of hashing functions

and p_f is false positive probability. A l-level cache counting Bloom filter is a Bloom filter with each counter pointing to the level corresponding to its counter value and with level l containing l buckets.The l-level cached counting Bloom filter architecture is depicted in Figure 4.4.

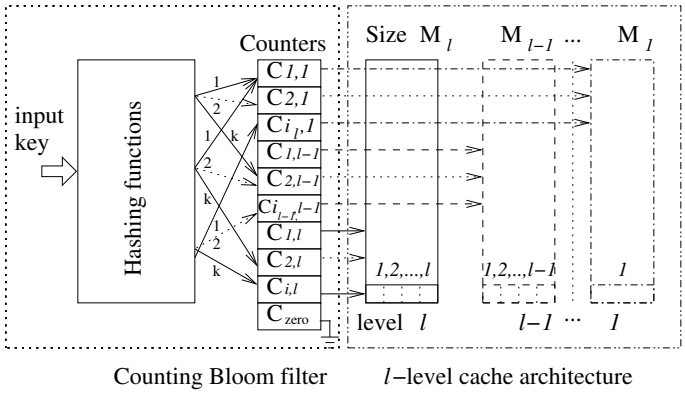


Figure 4.4: The l-level cached counting Bloom filter architecture.

In this figure, Ci_l, l represents the counter with the value 'l' pointing to location i_l within cache level 'l'. Therefore, the values of $C_{1,1}, ..., C_{i_1,1}$ are equal to 1, the values of $C_{1,l-1}, ..., C_{i_{l-1},l-1}$ are equal to $l-1$ and the values of $C_{1,l}, ..., C_{i,l}$ are equal to l. C_{zero} shows the counters with value 0 and does not point to any bucket in the cache memory. These counters are represented by C_{zero}. M_l represents the size of the cache memory in level l. From the Figure 4.4, the number of accesses in l-level CCBF is equal to summation of accesses in each level as follows:

$$N_{Total\ l-CCBF} \quad = \quad (N_1 + ... + N_i + ... + N_l) \tag{4.3}$$

In this equation, N_i represents the number of accesses in level i and $N_{Total\ l-CCBF}$ shows the total number of accesses in the l-level CCBF. Based on definition of the CCBF, the size of a bucket in level i is equal to i. Therefore, in each access, i items can be transferred. Consequently, the number of accesses depends on the number of levels that means the utilization of multi-level cached counting Bloom filter decreases the number of accesses. The number of accesses in level i is equal to number of the buckets in this level. To calculate the number of buckets, the size of level i is divided by size of the bucket in this level. From Equations 2.8 and 4.3, the expected number of accesses in CCBF is

presented as follows:

$$\begin{aligned} N_{Total\ l-CCBF} &= A\left(p(j=1) + \tfrac{p(j=2)}{2} + ... + \tfrac{p(j=l)}{l}\right) \\ &= A\begin{pmatrix} A \\ 1 \end{pmatrix}\left(\tfrac{1}{m}\right)\left(1-\tfrac{1}{m}\right)^{A-1} + ... \\ &+ A\begin{pmatrix} A \\ l \end{pmatrix}\left(\tfrac{1}{l}\right)\left(\tfrac{1}{m}\right)^{l}\left(1-\tfrac{1}{m}\right)^{A-l} \\ & (with\ A = nk(1+p_f)) \end{aligned} \tag{4.4}$$

In Equation (4.4), $p(j=l)$ shows the probability that a counter incitements l time. $A\left(\dfrac{p(j=l)}{l}\right)$ represents the number of expected accesses in level l for a counting Bloom filter with n items and k hashing functions. To make simplify equations above, Equation (4.5) can be utilized:

$$\left(1-\frac{1}{m}\right)^{nk} \cong e^{\frac{-nk}{m}} \tag{4.5}$$

Using Equation (4.5), we can rewrite the Equation (4.4) as follows:

$$\begin{aligned} N_{Total\ l-CCBF} &= A\left(p(j=1) + \tfrac{p(j=2)}{2} + ... + \tfrac{p(j=l)}{l}\right) \\ &= Ae^{\frac{-A}{m}}\left(\left(\tfrac{1}{m}\right)\begin{pmatrix} A \\ 1 \end{pmatrix} + ... + \begin{pmatrix} A \\ l \end{pmatrix}\left(\tfrac{1}{l}\right)\left(\tfrac{1}{m}\right)^{l}\right) \\ &\cong Ae^{\frac{-A}{m}}\left(\sum_{i=1}^{l}\tfrac{1}{ii!}\left(\tfrac{A}{m}\right)^{i}\right) \\ & (with\ A = nk(1+p_f)) \end{aligned} \tag{4.6}$$

If we assume that $\frac{m}{n} = c$ then we can rewrite Equation (4.6) as follows:

$$N_{Total\ l-CCBF} = nk(1+p_f)e^{\frac{-k}{c}(1+p_f)}\left(\sum_{i=1}^{l}\tfrac{1}{ii!}\left(\tfrac{k(1+p_f)}{c}\right)^{i}\right) \tag{4.7}$$

After the normalization to $nk(1+p_f)$ number of accesses is expressed as function of c and k as follows:

$$N_{Total\ l-CCBF} = e^{\frac{-k(1+p_f)}{c}}\left(\sum_{i=1}^{l}\tfrac{1}{ii!}\left(\tfrac{k(1+p_f)}{c}\right)^{i}\right) \tag{4.8}$$

An l-level CCBF needs buckets with length l, that this means the CCBF must support to transfer a bucket with l item in one I/O operation. In other words, the designing of l level CCBF is impractical. Therefore, we propose a CCBF with limited number of levels. Our observations from Equation (4.2) and the graph depicted in Figure 4.3 show that the counter values are not likely to be larger than 3. This means a 3-level CCBF is more beneficial than l-level CCBF. More

precisely, levels 1 and 2 store the elements for the counters with values 1 and 2, respectively. Level 3 stores the elements for counters with value 3 or larger. As the counters with values larger than 3 require more storage, the elements are stored over multiple rows in the third level of the CCBF (*segmentation*). The 3-level CCBF is depicted in Figure 4.5.

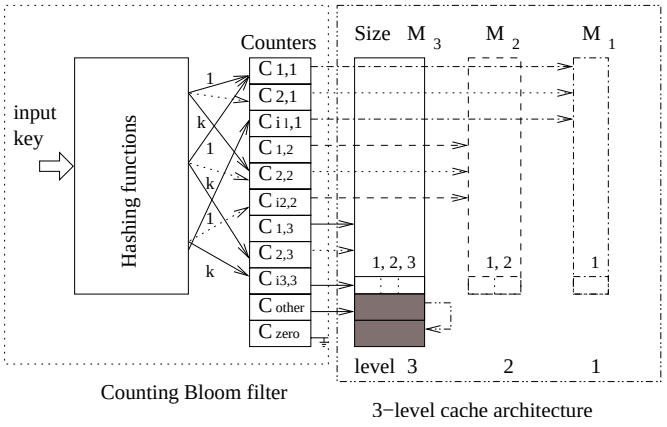


Figure 4.5: The 3-level cached counting Bloom filter architecture.

In Figure 4.5, the values of $C_{1,1}, ..., C_{i_1,1}$ are equal to 1, the values of $C_{1,2}, ..., C_{i_2,2}$ are equal to 2 and the values of $C_{1,3}, ..., C_{i_3,3}$ are equal to 3. C_{other} represents the counters with values larger than three and, therefore, they point to a storage within level 3 of the CCBF. Figure 4.5 highlights the mentioned segmentation. C_{zero} represents the counters with their value being zero. In the following, we analyze the effects of the items with counter values larger than three. The number of accesses in a 3-level CCBF is equal to number of accesses in the levels 1, 2, and 3 combined. The number of accesses in third level of cache can be computed as a summation of the number of counters with value 3 and larger. Therefore, the number of accesses in a 3-level CCBF is presented as follows:

$$
\begin{aligned}
N_{Total\ 3-CCBF} &= A\left(p(j=1)+\tfrac{p(j=2)}{2}+\tfrac{p(j\geq 3)}{3}\right)\\
&= A\left(p(j=1)+\tfrac{p(j=2)}{2}+\tfrac{p(j=3)}{3}\right)\\
&+ \left\lceil\tfrac{4}{3}\right\rceil p(j=4)A+...+\left\lceil\tfrac{l}{3}\right\rceil p(j=l)A\\
&= A\begin{pmatrix}A\\1\end{pmatrix}\left(\tfrac{1}{m}\right)\left(1-\tfrac{1}{m}\right)^{A-1}\\
&+ A\begin{pmatrix}A\\2\end{pmatrix}\left(\tfrac{1}{2}\right)\left(\tfrac{1}{m}\right)^{2}\left(1-\tfrac{1}{m}\right)^{A-2}\\
&+ A\begin{pmatrix}A\\3\end{pmatrix}\left(\tfrac{1}{3}\right)\left(\tfrac{1}{m}\right)^{3}\left(1-\tfrac{1}{m}\right)^{A-3}\\
&+ \textstyle\sum_{i=4}^{l}\left\lceil\frac{i}{3}\right\rceil\begin{pmatrix}A\\i\end{pmatrix}\left(\tfrac{1}{m}\right)^{i}\left(1-\tfrac{1}{m}\right)^{A-i}A\\
&\quad (with\ A=nk(1+p_f))
\end{aligned}
\tag{4.9}
$$

In Equation (4.9), $N_{Total\ 3-CCBF}$ shows total number of accesses in 3-level CCBF. Using Equation (4.5), the Equation (4.9) can be rewritten as follows:

$$
\begin{aligned}
N_{Total\ 3-CCBF} &\cong Ae^{\frac{-A}{m}}\left(\tfrac{A}{m}+\tfrac{1}{2*2!}(\tfrac{A}{m})^2+\tfrac{1}{3*3!}(\tfrac{A}{m})^3\right)\\
&+ \textstyle\sum_{i=4}^{l}\left\lceil\frac{i}{3}\right\rceil\frac{1}{i!}e^{\frac{-A}{m}}(\tfrac{A}{m})^{i}A\\
&\quad (with\ A=nk(1+p_f))
\end{aligned}
\tag{4.10}
$$

Numerical computations using Maple $v.12.0$ show that Equation (4.10) can be a good approximation for Equation (4.9). Therefore, we can write the number of accesses in a 3-level CCBF as follows:

$$
\begin{aligned}
N_{Total\ 3-CCBF} &\cong e^{\frac{-nk(1+p_f)}{m}}\left(\tfrac{nk(1+p_f)}{m}+\tfrac{1}{2*2!}(\tfrac{nk(1+p_f)}{m})^2+\tfrac{1}{3*3!}(\tfrac{nk(1+p_f)}{m})^3\right)\\
&+ \textstyle\sum_{i=4}^{l}\left\lceil\frac{i}{3}\right\rceil\frac{1}{i!}e^{\frac{-nk(1+p_f)}{m}}\left(\tfrac{nk(1+p_f)}{m}\right)^{i}
\end{aligned}
\tag{4.11}
$$

After substitution of $\frac{m}{n}$ with c the number of accesses in the 3-level CCBF is written as follows:

$$
\begin{aligned}
N_{Total\ 3-CCBF} &\cong e^{\frac{-k(1+p_f)}{c}}\left(\tfrac{k(1+p_f)}{c}+\tfrac{1}{2*2!}(\tfrac{k(1+p_f)}{c})^2+\tfrac{1}{3*3!}(\tfrac{k(1+p_f)}{c})^3\right)\\
&+ \textstyle\sum_{i=4}^{l}\left\lceil\frac{i}{3}\right\rceil\frac{1}{i!}e^{\frac{-k(1+p_f)}{c}}\left(\tfrac{k(1+p_f)}{c}\right)^{i}
\end{aligned}
\tag{4.12}
$$

In the following, we evaluate the size of the different cache levels in the CCBF architecture. In short, the size of each cache level is equal to the multiplication of nk and the probability of each counter value in the CCBF. The size of each cache level in l-level CCBF is expressed as follows:

$$
\begin{aligned}
Size_{j\ l-CCBF} &= nkjp\,(c(i)=(level\ number))\\
&= nkjp\,(c(i)=j)\\
&= nkj\begin{pmatrix}nk\\j\end{pmatrix}\left(\tfrac{1}{m}\right)^{j}\left(1-\tfrac{1}{m}\right)^{nk-j}
\end{aligned}
\tag{4.13}
$$

In Equation (4.13), j is level number, and $Size_{jl-CCBF}$ shows the size of level j in l-level CCBF. Using Equation (4.3) and Equation (4.5), we can rewrite Equation (4.13) as follows:

$$\begin{array}{rcl} Size_{j\ l-CCBF} & \cong & nkje^{\frac{-k}{c}}\left(\frac{1}{j!}\left(\frac{k}{c}\right)^{j}\right) \\ & & where\ j\ is\ level\ number \end{array} \tag{4.14}$$

Using Equation (4.14), the total size of the l-level CCBF cache after normalization to nk (size of a standard Bloom filter) is:

$$\begin{array}{rcl} Size_{Total\ l-CCBF} & = & nkj\sum_{j=1}^{l}(p(c(i)=j) \\ & \cong & e^{\frac{-k}{c}}\left(\sum_{j=1}^{l}\frac{j}{j!}\left(\frac{k}{c}\right)^{j}\right) \end{array} \tag{4.15}$$

In Equation (4.15), $Size_{Totall-CCBF}$ shows the total size of l-level CCBF. Applying this equation to the 3-level CCBF case, results in the following sizes of the 3 levels (keeping in mind 4-bit counter, the with l being 16):

$$\begin{array}{rcl} Size_{1\ 3-CCBF} & = & e^{\frac{-k}{c}}\frac{k}{c} \\ Size_{2\ 3-CCBF} & = & e^{\frac{-k}{c}}\left(\frac{k}{c}\right)^{2} \\ Size_{3\ 3-CCBF} & = & e^{\frac{-k}{c}}\left(\sum_{j=3}^{l}\frac{j}{j!}\left(\frac{k}{c}\right)^{j}\right) \end{array} \tag{4.16}$$

4.1.3 Hashing Functions

Several kinds of hashing functions are utilized in packet classification: additive, rotative, bit extraction, XOR-based, mixed, and universal hashing functions [81], [85]. In additive hashing functions, the hash value is constructed by traversing through the data and continually incrementing an initial value by a calculated value relative to an element within the data. The calculation performed on the element value is usually in the form of a multiplication by a prime number. In rotative hashing functions, every element in the data string is used to construct the hash value, but unlike additive hashing the values are put through a process of bitwise shifting. In the bit extraction method, the hashing function entails selecting j bits out of the i bits of the key. In XOR-based hashing functions, the i-bit key is partitioned into j-bit segments. The segments are exclusive-ORed to produce a hash address. In the mixed method, the hashing functions utilize any or all of the mentioned techniques. It obvious that the performance of these functions dependent on the key set [81], [85]. These hashing functions take longer to execute compared to the earlier mentioned functions that only utilize bitwise logical operations.

A solution to achieve a hashing function that is independent from the key set is by utilizing a class of universal hashing functions that exploits bitwise logical

operations in their definition. Let H represent a class of functions with input set A and output set B. H is said to be universal if for all x, y in A, no pair of distinct keys collide under more than $(1/|B|)$th of the functions where $|B|$ denotes size of B [29]. A special class of universal hashing functions is called $H3$ hashing functions [29], [85]. The $H3$ class of hashing functions are defined as follows: Let $A = 0, 1, 2, ..., 2^i - 1$ be the key space, $B = 0, 1, 2, ..., 2^j - 1$ be the address space, i denotes the number of bits in the key, j denotes the number of bits in address, Q denotes the set of $i \times j$ boolean matrices, and I represents the given key set, $I = x_1, x_2, ..., x_n$, $I \subset A$. For a given $q \in Q$ and $x \in A$, let $q(k)$ be the kth row of the matrix q and x_k be the kth bit of x. The hashing function $h_q(x) : A \rightarrow B$ is defined as follows:

$$h_q(x) = x_1.q(1) \bigoplus x_2.q(2) \bigoplus ... \bigoplus x_i.q(i) \tag{4.17}$$

where . denotes the binary AND operation and $\bigoplus$ the XOR operation. The hashing function from this class can be easily implemented in hardware. The hardware stores the $i \times j$ boolean matrix that can be organized in a bank of registers [85] where the boolean matrices can be generated in software and then loaded into the bank of registers. We present an example of $H3$ hashing function in the following.

Example: Let i be 4 and j be 3, then the address space is $A = 0..15$ and the key space is $B = 0..7$, we randomly select an 4×3 matrix q as follows:

$$q = \begin{pmatrix} 010 \\ 110 \\ 100 \\ 001 \end{pmatrix}$$

Then the hash address for key 13 and 10 are:

$$\begin{aligned} h_q(13) &= h_q(1101) = q(1) \bigoplus q(2) \bigoplus q(4) \\ &= 010 \bigoplus 110 \bigoplus 001 = 101 \\ &= 5 \end{aligned}$$

$$\begin{aligned} h_q(10) &= h_q(1010) = q(1) \bigoplus q(3) \\ &= 010 \bigoplus 100 = 110 \\ &= 6 \end{aligned}$$

Based on the tuple space representation for the rule-set database and IP packets, the size of the input key is 88 bits long (32 bit source IP address, 32 bit

destination IP address, 8 bit Range-ID for source port, 8 bit Range-ID for destination port and 8 bit protocol field). The maximum size of the tuple or address space is assumed to be 2^{16} rules for 16 bit address. Therefore, $Q_{88\times16}$ denotes a set of matrices to define the $H3$ class of hashing functions in the tuple space packet classification algorithm [11].

4.1.4 Packet Classifier Architecture Using Bloom Filter

In this section, we present a software packet classifier using the pruned counting Bloom filter. The two main parts of our architecture are: rule hash table constructor and packet search. The architecture is depicted in Figure 4.6.

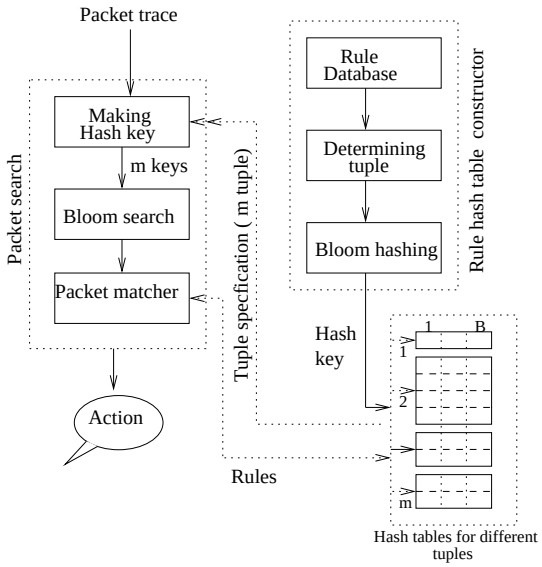


Figure 4.6: The architecture of classifier using pruned counting Bloom filter and tuple space.

Rule hash table constructor component reads the rules from a rule set database and extracts the rule specification to determine the related tuple that the rule belongs to. After determining the tuple, the rule should be hashed, the next procedure is making a hash key using the Bloom filter and finally store the rules in the hash table. In this process, different hash tables with unequal size are created, since each hash table correspond to a single tuple and each tuple

included different rules and tuple specifications. Usually more than half of the rules belong to two tuples, and this is depicted on the right side of Figure 4.6. In this figure, m is the number of tuples and B shows the size of the buckets. Packet search component processes the incoming packets to find matching rules in the hash tables corresponding to tuples. Therefore, for each incoming packet, a hash key is extracted based on each tuple specification. Subsequently, m hash keys are used to access the m hash tables (after hashing) to determine whether matching rules can be found. The accessing of the hash table can be performed in a serial or parallel manner. Finally, the actual packet is checked against the found rules in the packet matcher. For each packet, the number of hashing operations are equal to the number of tuples in the system or the number of distinct hash tables, thus the number of access in sequential search process per packet is equal to the number of tuples.

4.1.5 Packet Classifier Architecture Using CCBF

In this section, we present the software packet classifier architecture that utilizes a CCBF to store and retrieve the rules and incoming packets. The packet classifier architecture includes three components as follows: rule hash table constructor, packet search and CCBF unit. The software packet classifier architecture is depicted in Figure 4.7.

Rule hash table constructor

This component reads the rules from a rule-set database and extracts the rule specification to determine the related tuple that the rule belongs to. After determining the tuple, the rule should be hashed, the next procedure is making a hash key using the Bloom filter and finally stores the rules in the hash table. In this process, different hash tables with unequal size are created by CCBF unit, since each hash table correspond to a single tuple and each tuple included different rules and tuple specifications. Usually more than half of the rules belong to two tuples.

Packet search

This component of the classifier processes the incoming packets to find matching rules in the hash tables corresponding to tuples. Therefore, for each incoming packet, a hash key is extracted based on each tuple specification. Subsequently, m hash keys are used to access the m hash tables (after hashing) to determine

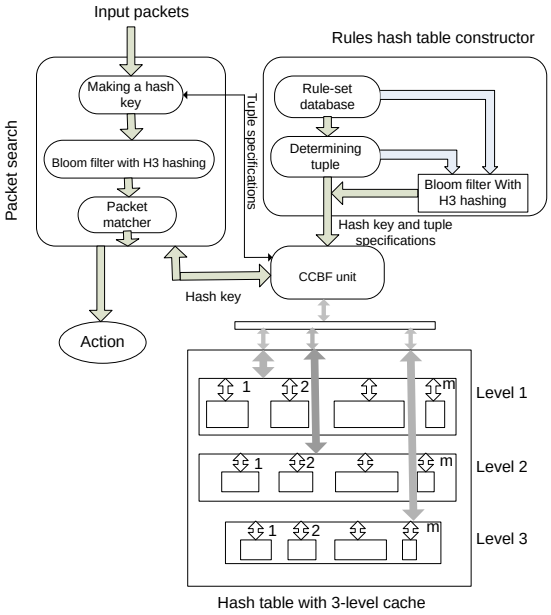


Figure 4.7: The architecture of software packet classifier using CCBF.

whether matching rules can be found. The accessing of the hash table can be performed in a serial or parallel manner. Consequently, the actual packet is checked against the found rules in the packet matcher. For each packet, the number of hashing operations are equal to the number of tuples in the system or the number of distinct hash tables, thus the number of access in sequential search process per packet is equal to the number of tuples.

CCBF unit

An important part of the packet classifier architecture that is utilized by the rule hash table constructor and packet search components is the CCBF unit. The CCBF unit manages the counters and different cache levels. When an item (rule) is inserted the related counters are changed, therefore, the cache level of these items should be updated and the related buckets should move to the new cache level. All counters are initialized to zero, therefore, after the insertion some items, these counters should be updated by at lease one subsequently, these items are stored in the level one and after the incrementing counters, these items move to higher level. The content of the cache levels in the CCBF unit are modified by the rule hash table constructor component but will only be

inspected by the packet search component. The CCBF unit manages the I/O operations to/from cache levels. In the first and second cache levels with the counter values 1 and 2 the I/O buffer size (bucket) is set to 1 and 2 and in the third cache level the I/O buffer size is set to three. The CCBF unit segments the buffer pointed to by counters that are larger than 3.

4.2 A Memory Optimization Approach for Bloom Filters using an Additional Hashing Function

In this section, we introduce a technique to eliminate the redundant items in the standard and counting Bloom filters. In the standard and counting Bloom filters, over an input item, k hashing functions are computed and the input item is stored in the locations pointed to by the generated addresses produced by the hashing functions. Therefore, in the standard and counting Bloom filters we keep k copies of each item but only one copy is accessed and the other $(k-1)$ copies of items are never accessed. As we discussed in Chapter 2.3.2, to optimize memory in the standard and counting Bloom filters, a caching policy or pruning technique can be used, respectively. The mentioned caching policy exploits the Bloom filter properties to decrease the number of memory accesses. The pruning technique reduces the memory utilization but has some limitations as: high processing time due to incremental update, reconsideration of all items when inserting items, and limited use as it works only in conjunction with counting Bloom filter and thereby reducing its applicability.

We propose memory optimization technique using an additional hashing function and call Bloom filter using additional hashing function (BFAH). Our proposal selects one of the generated addresses by the k hashing functions in the Bloom filters. The utilization of an additional hashing function has the following advantages: decreasing memory redundancy in comparison to standard Bloom filters, and increasing the performance in comparison to standard Bloom filters, and its simplicity to implement in hardware. Subsequently, we analyze different performance metrics for BFAH, pruned counting Bloom filter (PCBF) and standard Bloom filter. The performance metrics that we consider are: average bucket size, maximum search length, and the number of collisions.

4.2.1 The BFAH Architecture and Concept

In the standard and counting Bloom filters, each item is stored k times but only one copy is accessed. For the counting Bloom filter, a pruning procedure

was proposed to minimize the memory utilization. Still, in the standard and counting Bloom filters, the following questions should be addressed: How are redundant items eliminated? How many empty slots without any item can be found in the bit-array or array of counters? What is the maximum number of accesses to fetch different items? Therefore, we utilize an additional hashing function in the standard and counting Bloom filters to address these questions. The proposed solution has the following features:

- Minimization of memory redundancy: In the BFAH each item is stored only once.
- The distribution of the incoming items is more randomized in comparison to other Bloom filters since in this procedure, the selection from the generated addresses by hashing function is performed by a hashing technique that assists in distributing incoming items uniformly.
- It can be applied to the all Bloom filter types. This is because, it works based on the bit-array of Bloom filter.
- In the pruning technique the searching for an incoming item, requires inspection of k counters in counting Bloom filter while in the BFAH a single hash value is computed.

The architecture of the BFAH is depicted in Figure 4.8.

In Figure 4.8, an incoming item is hashed by k hashing functions and the corresponding bits are set (or counters are incremented in the counting Bloom filter) (see block A in Figure 4.8). Subsequently, one of the generated addresses by the k hashing functions is selected by another hashing function. This address is used to store the item in question. The additional hashing function receives the incoming key and selects one of k received addresses to store in the memory. The output of the additional hashing function is a number that represents the index of one of k addresses pointed to by hashing functions (see block B in Figure 4.8). The generated addresses by the k hashing functions (block A in Figure 4.8) and generated number by the additional hashing function is processed by the address selector unit in block A. This component selects an address among the incoming addresses. It should be noted, that the additional hashing function works alongside the others k hashing functions. Therefore, the selected address by the additional hashing function and generated addresses by k hashing functions are available at the same time in the address selector unit. Additionally, in the pruned counting Bloom filter, the counters should be

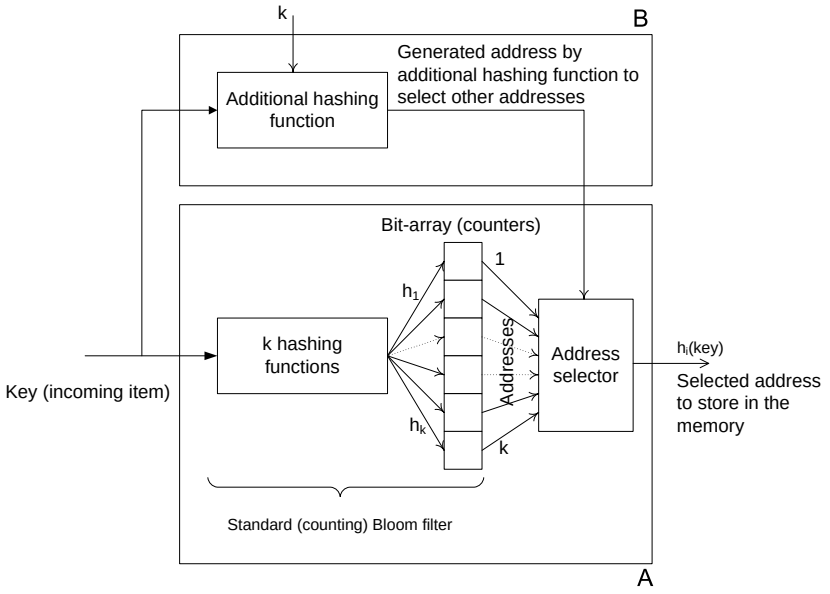


Figure 4.8: A Bloom filter architecture with an additional hashing function.

searched to find the minimum counter but in this approach only one hashing key by the additional hashing function is calculated.

Figure 4.9, depicts a standard/counting Bloom filter for four items. In this Figure, bit-array and counters show standard and counting Bloom filters, respectively.

Figure 4.10 depicts an example of how the BFAH operates using the Bloom filter in Figure 4.9 as starting point. In this example, we utilize the divide hashing function "$key \ mod \ k$" as additional hashing function where k represents the number of hashing functions and index of incoming items is supposed as input key. This is due to the simplicity of the hashing function in the example.

Figure 4.10 (a), depicts a Bloom filter before any insertion therefore the values of the counters or bits in the bit-array are zero. In the first step, rule $R0$ is hashed to addresses 0, 2, and 4 using $h0$, $h1$, and $h2$ hashing functions (see Figure 4.10 (b)). The value of the counters are incremented (or bit-array is set) and then the generated addresses by hashing functions should be selected to store the rule $R0$ in the hashed address. Therefore, an additional hashing function selects one address out of k generated addresses. The additional hashing function selects

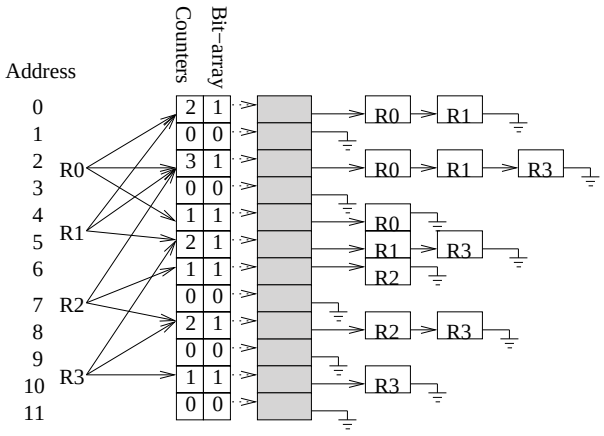


Figure 4.9: A standard/counting Bloom filter architecture for four items.

address 0 that was generated by hashing function $h0$ and $R0$ is stored in the address 0. This is because the index of $R0$ that is '0', is divided to 3 ($0 \; mod \; 3 = 0$) and, finally address 0 is selected to store the rule $R0$. In this example number of hashing functions is 3.

Subsequently, $R1$ is hashed to addresses 0, 2, and 5 (see Figure 4.10 (c)).The values of counters are incremented (bit-array is set) to 2, 2, and 1, respectively. Using the additional hashing function "$key \; mod \; k$", address 2 that was generated by hash function $h1$ is selected to store the $R1$. As depicted in Figure 4.10 (d), $R2$ is hashed to addresses 2, 6, and 8 and address 8 that generated by $h2$ is selected to store rule $R2$. Finally, rule $R3$ after the hashing to addresses 5, 8 and 10 is stored at address 5.

4.2.2 The BFAH Architecture Analysis

We analyze the standard, pruned counting Bloom filters and the BFAH. In this analysis, the average bucket size, the maximum search length and the number of collisions are investigated.

Analysis of average bucket size

The average bucket size is defined as the total number of stored items in the Bloom filter divided by the number of non-empty entries (buckets). In the

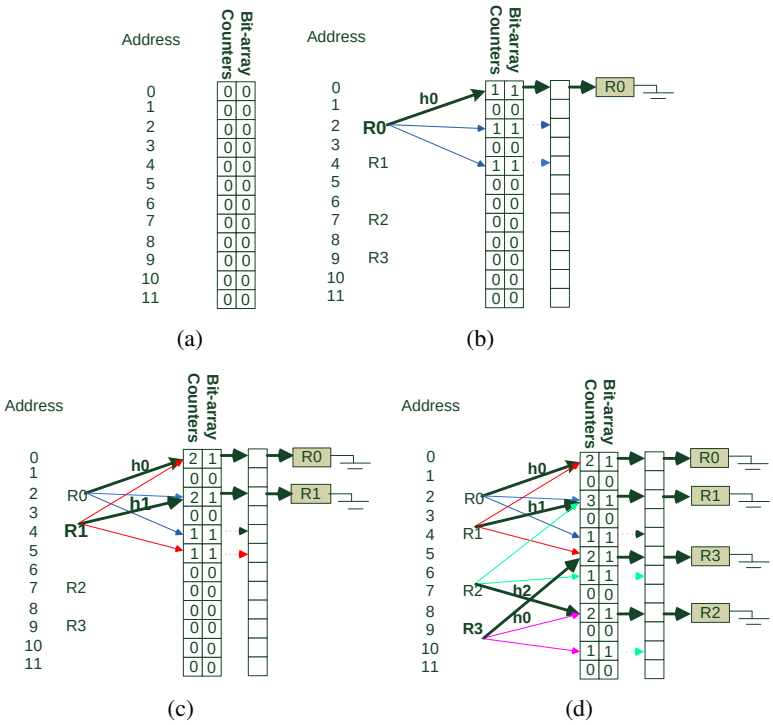


Figure 4.10: The hash table architecture using a Bloom filter with an additional hashing function (a) An empty Bloom filter. (b) The Bloom filter after the insertion of rule $R0$ (c) The Bloom filter after the insertion of rules $R0$ and $R1$ (d) Final Bloom filter after the insertion of four rules.

standard Bloom filter with n items and nk insertions, the probability a bucket receives exactly j insertions is expressed as:

$$p\left(b(i)=j\right)=\left(\begin{array}{c} nk \\ j \end{array}\right)\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j} \tag{4.18}$$

In Equation (4.18), $b(i)$ represents the number of items in $bucket(i)$. In order to calculate the probability of an empty bucket, j is set to 0. Therefore, the probability of non-empty buckets is equal to 1 minus the probability of the empty buckets. Consequently, the probability of a non-empty bucket is calculated as follows:

$$p(non-empty\,buckets)=\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \tag{4.19}$$

From Equation (4.19), the expected number of the non-empty buckets is calculated as follows:

$$E(Number\ of\ non-empty\ buckets) = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \quad (4.20)$$

In a standard Bloom filter, the average bucket size is as follows:

$$Average\ bucket\ size\ in\ BF = \frac{Number\ of\ stored\ items}{Number\ of\ non-empty\ buckets} = \frac{nk}{m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)} \quad (4.21)$$

Initially, in the BFAH, in the first step, for n items nk hash operations are performed using k hashing functions and the related bits in the bit-array are set. In a Bloom filter with m bits (bit-array size), n items and k hashing functions, $1/m$ represents the probability any one of the m bits set by a single hashing function operating on a single input item. $(1-1/m)$ is the probability that the bit is unset after a single hash value computation with a single item. The probability that a bit is still unset after all the items are hashed into the Bloom filter by using k independent hashing functions is $\left(1-\frac{1}{m}\right)^{kn}$. Therefore, the probability of set bits in the bit-array is $\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. Consequently, the expected number of set bits in the bit-array is calculated using $t = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects n out of t addresses that set in the previous step. Using Equation (4.20), the number of selected items by the additional hashing function occupy $t\left(1-\left(1-\frac{1}{t}\right)^{n}\right)$ addresses, where t is the number of set bits in the bit-array. Therefore, the average bucket size in the BFAH is calculated as follows:

$$Average\ bucket\ size\ in\ BFAH = \frac{n}{t\left(1-\left(1-\frac{1}{t}\right)^{n}\right)}\ with\ t = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \quad (4.22)$$

It should be noted, that n items are stored in the BFAH. Pruned counting Bloom filter can be assumed as a standard Bloom filter with one hashing function that selects n out of nk hashed items. Therefore, the average bucket size for pruned counting Bloom filter is computed using Equation (4.22) when $k = 1$. Equation (4.23), shows the average bucket size for pruned counting Bloom filter.

$$Average\ bucket\ size\ in\ pruned\ counting\ Bloom\ filter = \frac{n}{m\left(1-\left(1-\frac{1}{m}\right)^{n}\right)} \quad (4.23)$$

The average bucket size for standard, BFAH and pruned counting Bloom filters using Equations (4.21), (4.22) and (4.23) for the configuration $k = ln(2)m/n$ that generates minimum false positive probability is depicted in Figure 4.11.

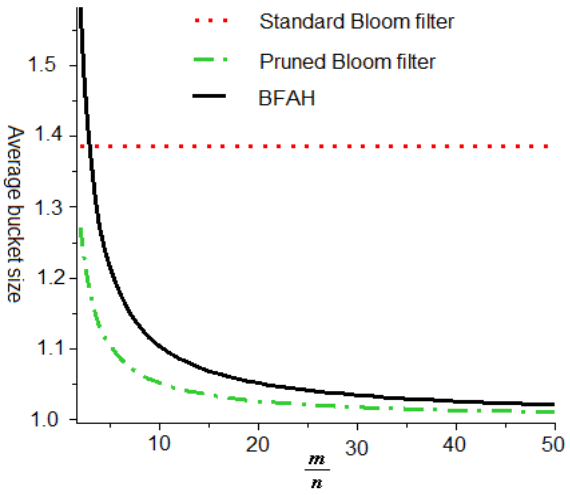


Figure 4.11: Average bucket size for standard, BFAH and pruned counting Bloom filters when $ln(2)k = m/n$).

Figure 4.11 depicts average bucket size in term of m/n (m is number of bits in the bit-array and n is the number of items). To generate minimum false positive probability $k = ln(2)m/n$. From Figure 4.11, we can observe that in a standard Bloom filter average bucket size with minimum false positive probability is a constant value 1.38. The shortest average bucket size belongs to the pruned counting Bloom filter. The graph in Figure 4.11 shows that BFAH architecture has average bucket size shorter than standard Bloom filter and longer than pruned counting Bloom filter. Additionally, from Figure 4.11, we can observe that when the value of m/n is increased the average bucket size of BFAH and pruned counting Bloom filter is closed to each other. In this case, for $m/n = 50$ the difference of average bucket size between BFAH and pruned counting Bloom filter is less than 1%.

Maximum search length

The maximum search length is defined as a maximum number of items that are inserted in the buckets. It can be used as a worst case search to access an item. In the standard Bloom filter, the expected number of items which their buckets included j items is equal to average number of buckets multiplied by the item

per bucket Equation (4.18), therefore, it is calculated as follows:

$$E(Number\ of\ items\ which\ their\ buckets\ included\ j\ items) = nkj\left(\left(\begin{array}{c} nk \\ j \end{array}\right)\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j}\right) \tag{4.24}$$

In Equation (4.24), nk is the number of hashed items and j is the bucket size (number of items per bucket). In BFAH, in first step, all nk hashed items are hashed using k hashing functions and related bits in the bit-array are set. The number of set bits in the bit-array is calculated using $t = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects n out of t addresses that set in the previous step. Therefore, in a BFAH, the probability of a bucket with j items is represented as follows:

$$p(\ Bucket\ with\ j\ items) = \left(\begin{array}{c} n \\ j \end{array}\right)\left(\frac{1}{t}\right)^{j}\left(1-\frac{1}{t}\right)^{n-j} \tag{4.25}$$

Using Equation (4.25), the number of items with bucket size j in the BFAH is calculated as follows:

$$E(Number\ of\ items\ which\ \ bucket\ size = j) = nj\left(\left(\begin{array}{c} n \\ j \end{array}\right)\left(\frac{1}{t}\right)^{j}\left(1-\frac{1}{t}\right)^{n-j}\right) \tag{4.26}$$

For pruned counting Bloom filter, maximum search length is computed when in the standard Bloom filter the number of hashing functions is set to 1. Hence, the number of items with bucket size j is calculated as follows:

$$E(Number\ of\ items\ which\ \ bucket\ size = j) = nj\left(\left(\begin{array}{c} n \\ j \end{array}\right)\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{n-j}\right) \tag{4.27}$$

Using Equations (4.24), (4.26) and (4.28), the maximum search length for standard, pruned counting Bloom filters and BFAH when $k = ln(2)m/n$ is depicted in Figure 4.12.

From Figure 4.12, it can be observed that, the maximum search length of BFAH is shorter than standard Bloom filter and longer than pruned counting Bloom filter. As an example, for $m = 10000$, $n = 2000$ and $k = 4$, 120, 3, and 1 bucket received more than 3 items in the standard, BFAH, and pruned counting Bloom filters, respectively.

Number of collisions

A common problem in using hashing is collision which means the mapping of incoming items to the same hash table location. Consequently, when an

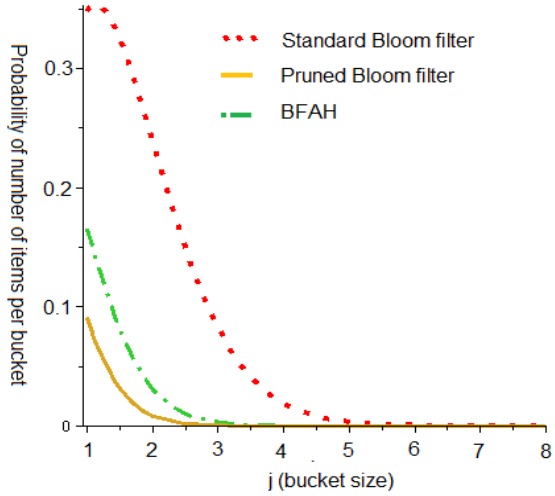


Figure 4.12: Maximum search length for standard, BFAH and pruned counting Bloom filters when $k = ln(2)m/n$.

incoming item is hashed to a hash table entry containing multiple items it must be matched to all these items resulting in a much longer processing time. In the standard Bloom filter, for each item, collision is detected when the number of hashed items to the related location is larger than the bucket size. Therefore, in the standard Bloom filter (using Equation (4.18)), the probability of number of collisions for i^{th} address in the bit-array is as follows:

$$p\,(col(i)\ with\ bucket\ size\ p) = \sum_{j=p}^{p_{max}-1}(j+1-p)\begin{pmatrix} nk \\ j+1 \end{pmatrix}\left(\tfrac{1}{m}\right)^{j+1}\left(1-\tfrac{1}{m}\right)^{nk-j-1} \tag{4.28}$$

In Equation (4.28), $col(i)$ shows the number of collisions for i^{th} address when the bucket size is set to p, while p_{max} shows the maximum bucket size. Using Equation (4.28), the expected number of collisions in the standard Bloom filter is as follows.

$$E\,(Number\ of\ collisions\ in\ the\ standard\ BF) = nk\sum_{j=p}^{p_{max}-1}(j+1-p)\begin{pmatrix} nk \\ j+1 \end{pmatrix}\left(\tfrac{1}{m}\right)^{j+1}\left(1-\tfrac{1}{m}\right)^{nk-j-1} \tag{4.29}$$

It is obvious that to calculate the number of hashed items to each address the value of p is set to 1. As we discussed previously, we can extend Equation (4.28)

for BFAH and pruned counting Bloom filters. In the BFAH, in first step, all nk hashed items are hashed using k hashing functions and related bits in the bit-array are set. The number of set bits in the bit-array is calculated using $t = m\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects n out of t addresses that set in the previous step. Therefore, in a BFAH (using Equation (4.28)), the expected number of collisions is presented as follows.

$$E\left(Number\ of\ collisions\ in\ the\ BFAH\right) = n\sum_{j=p}^{p_{max}-1}(j+1-p)\left(\begin{array}{c} n \\ j+1 \end{array}\right)\left(\frac{1}{t}\right)^{j+1}\left(1-\frac{1}{t}\right)^{n-j-1} \tag{4.30}$$

For pruned counting Bloom filter (PCBF), number of collisions is computed when in the standard Bloom filter the number of hashing functions is set to 1. Hence, the number of collisions is calculated as follows.

$$E\left(Number\ of\ collisions\ in\ PCBF\right) = n\sum_{j=p}^{p_{max}-1}(j+1-p)\left(\begin{array}{c} n \\ j+1 \end{array}\right)\left(\frac{1}{m}\right)^{j+1}\left(1-\frac{1}{m}\right)^{n-j-1} \tag{4.31}$$

Using Equations (4.29), (4.30) and (4.31), the number of collisions for the standard, pruned counting Bloom filters (PCBF) and the BFAH when $k = ln(2)m/n$ is depicted in Figure 4.13. In this figure, the number of collisions in the BFAH and pruned counting Bloom filters is normalized to n and in the standard Bloom filter is normalized to nk. In the BFAH and pruned counting Bloom filter number of the stored items is n while in the standard Bloom filter is nk. In addition, the value of p_{max} is set to 16. This is because, the analysis by Fan, et al. [43] for counting Bloom filter shows 4-bit counter is enough for most applications. In the other words, in all types of Bloom filters the probability for each address to receive more than 16 items is very small.

In Figure 4.13, we can observe that the number of collisions in BFAH and pruned counting Bloom filter is converged when the value of $\frac{m}{n}$ is increased. As an example, for $m = 20000$, $n = 1000$ and $k = 14$ the number of collisions are 1, 4 and 2509 for pruned counting Bloom filter, BFAH and standard Bloom filter. It should be noted that in this example, 1000 items are stored in the BFAH and pruned counting Bloom filter and 14000 items are stored in the standard Bloom filter

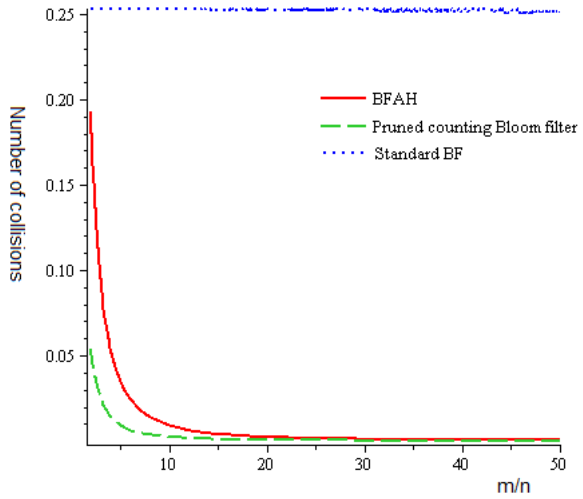


Figure 4.13: Number of collisions for standard, BFAH and pruned counting Bloom filters when $k = ln(2)m/n$.

4.3 k-stage Pipelined Bloom Filter for Packet Classification

Demand for high-performance network applications and devices is constantly driving the need for low-power solutions at the chip, system, and algorithm levels. These low-power management solutions are key in the industry's continuing quest to become smaller, cheaper, and high-performance. Due to the large-scale integration and high speed circuitry, network processors deployed in typical network equipment can consume more power than other components. As a key attached component to the network processor, the packet classifier must definitely be designed with power efficiency in mind. In this section,we present a k-stage pipelined Bloom filter architecture and the analysis of power consumption [16].

4.3.1 Power Model for Standard Bloom Filter

A Bloom filter consists of a set of hashing functions and a bit-array to lookup. Therefore, the total power consumed by the Bloom filter is a summation of power consumed by the hashing functions and the bit-array lookup. The archi-

tecture of a standard Bloom filter is depicted in Figure 4.14.

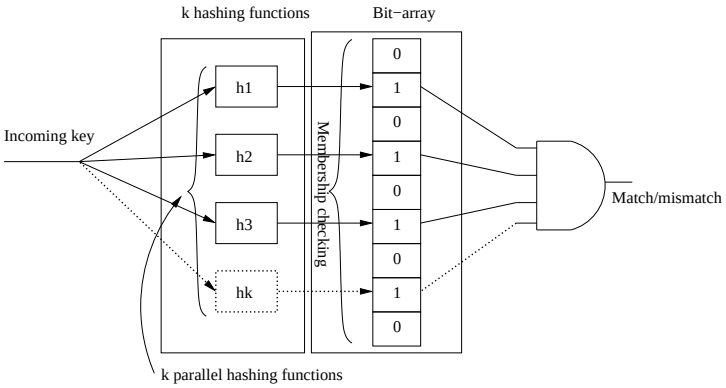


Figure 4.14: A standard Bloom filter with k hashing functions.

In Figure 4.14, a Bloom filter includes k hashing functions that are working simultaneously. In a Bloom filter, all k-bits pointed by hashing functions are set in the bit-array (m bits) in the programming stage and are checked in the membership checking stage. Universal hashing functions are generally utilized in the Bloom filter. We utilized a class of universal hashing functions that is called H3 hashing functions[29], [85], [10]. A standard Bloom filter utilizes k hashing functions in order to make a decision on the input. Hence, the power consumed by a standard Bloom filter depicted in Figure 4.14 is a summation of the power consumed by the hashing functions that is represented by Po_{hash}, and the power consumed by bit-array lookup operations that is represented by Po_{lookup}, plus the power consumed by k-input *'and'* operation that is represented by $Po_{Gate_{And_k}}$.

$$Po_{BF_{Standard}} = \sum_{i=1}^{k} (Po_{hash_i} + Po_{Lookup}) + Po_{Gate_{And_k}} \tag{4.32}$$

In Equation (4.32), the power consumed by all hashing functions is equal, therefore, Equation (4.32) is rewritten as follows:

$$Po_{BF_{Standard}} = k\,(Po_{hash} + Po_{Lookup}) + Po_{Gate_{And_k}} \tag{4.33}$$

4.3.2 k-stage Pipelined Bloom Filter

In this section, we present the concept of the k-stage pipelined Bloom filter and its analysis. The k-stage pipelined Bloom filter is a Bloom filter that implements its hashing functions in a pipelined manner. A k-stage pipelined Bloom filter architecture is depicted in Figure 4.15.

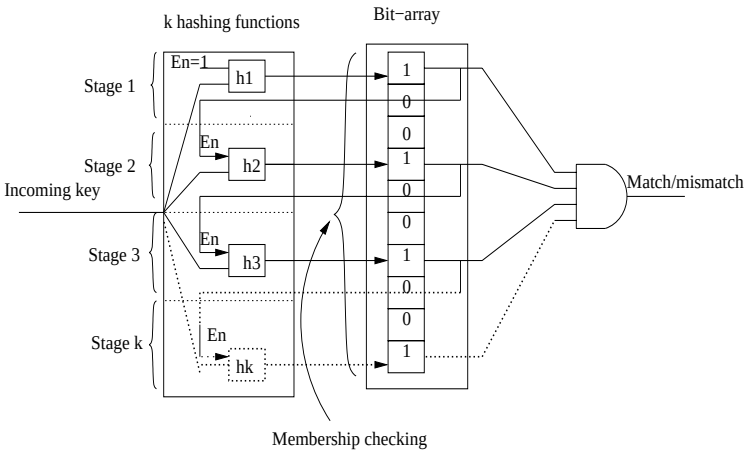


Figure 4.15: k-stage pipelined Bloom filter architecture.

Basically, a k-stage pipelined Bloom filter as depicted in Figure 4.15 consists of k groups of hashing functions. Each stage always computes the hash values and the next stage only compute the hash values if in the previous stage there is a match between the input item and the bit-array sought. In Figure 4.15, *'En'* represents matching in the previous stage and enables the next stage of the pipeline. The *'En'* signal adds a delay to the pipelined Bloom filter that causes to increase the power consumed by the hashing function. The power consumed due to the delay in compared to the power consumed by hashing functions in the standard Bloom filter is negligible. The advantage of using a k-stage pipelined Bloom filter is that if the current stage produces a mismatch, there is no need to use the next stages in order to decide whether the input item is a member of the bit-array. At worst, it will operate like a standard Bloom filter, which utilizes all of the hashing functions before making a decision on the type of the input. The k-stage pipelined Bloom filter is utilized in membership checking stage, since in the programming stage all hashing functions are utilized and pipeline stages are permanently full. In a Bloom filter with m bits (bit-array size), n items and k hashing functions, $1/m$ represents the probability that any one of the m bits

set by a single hashing function operating on a single input item. $(1 - 1/m)$ is the probability that the bit is unset after a single hash value computation with a single item [16]. The probability that a bit is still unset after all the items are programmed into the pipelined Bloom filter by using k independent hashing functions is as follows:

$$P_{unset\ bit} = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} \tag{4.34}$$

Consequently, the probability that any one of the bits is set is as follows:

$$p_{set} = (1 - p_{unset\ bit}) = 1 - e^{\frac{-kn}{m}} \tag{4.35}$$

From Figure 4.15, we can write the power consumed by the pipelined Bloom filter as follows:

$$\begin{aligned}
Po_{BF_{pipelined}} &= Po_{hash_{h_1}} + Po_{lookup_{h_1}} \\
&+ p_{set_{h_1}} \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) \\
&+ p_{set_{h_1}} p_{set_{h_2}} \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) + ... \\
&+ p_{set_{h_1}} ... p_{set_{h_{k-1}}} \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) + Po_{Gate_{and_k}} \\
&= Po_{hash_{h_1}} + Po_{lookup_{h_1}} \\
&+ A \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) \\
&+ A^2 \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) + ... \\
&+ A^{k-1} \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) + Po_{Gate_{and_k}} \\
&= \textstyle\sum_{i=1}^{k} A^{i-1} \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) + Po_{Gate_{And_k}} \\
&\\
&with\ A = \left(1 - e^{\frac{-kn}{m}}\right)
\end{aligned} \tag{4.36}$$

In Equation (4.36), $p_{set_{h_i}}$ represents the probability to set the bit pointed to by hashing function with index i. The power consumed by the pipelined Bloom filter is represented as follows:

$$\begin{aligned}
Po_{BF_{pipelined}} &= \textstyle\sum_{i=1}^{k} \left(1 - e^{\frac{-kn}{m}}\right)^{i-1} \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) \\
&+ Po_{Gate_{And_k}}
\end{aligned} \tag{4.37}$$

From Equations (4.33) and (4.37), we can observe that difference between these equations is related to their coefficients, the coefficient of Equation (4.33) is k and coefficient of Equation (4.37) is $\sum_{i=1}^{k}\left(1-e^{\frac{-kn}{m}}\right)^{i-1}$. In other words, the power consumed by the k-stage pipelined Bloom filter is less than the power consumed by the standard Bloom filter if $\sum_{i=1}^{k}\left(1-e^{\frac{-kn}{m}}\right)^{i-1} < k$ or $\frac{\sum_{i=1}^{k}\left(1-e^{\frac{-kn}{m}}\right)^{i-1}}{k} \leq 1$.

$$Coefficient\ rate = \frac{\sum_{i=1}^{k}\left(1-e^{\frac{-kn}{m}}\right)^{i-1}}{k} \tag{4.38}$$

It should be noted that Equation (4.38) represents the power consumed by a k-stage pipelined Bloom filter that is normalized to the power consumed by a standard Bloom filter.

4.3.3 4-stage pipeline Bloom filter

The architecture of a 4-stage pipelined Bloom filter is depicted in Figure 4.16.

In this figure, we can observe that the first three stages that are most frequently used include only one hashing function and last stage includes other $k-3$ hashing functions. This fact decreases power consumption in comparison to a standard Bloom filter and pipeline latency in comparison to k-stage pipelined Bloom filter architecture. Similar to the power model of the k-stage pipelined Bloom filter, the power model for the 4-stage pipelined Bloom filter is as

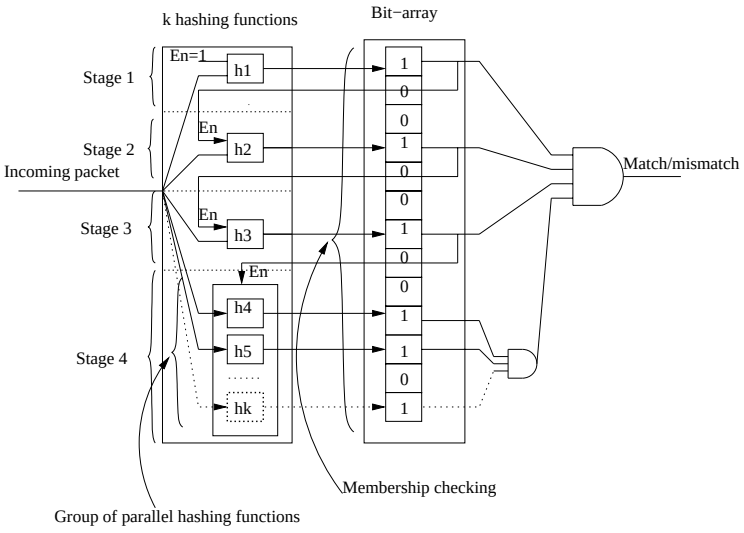


Figure 4.16: Our 4-stage pipelined Bloom filter architecture where the first three stages contains one hashing function and the forth stage contains $k-3$ hashing functions that operate in a parallel manner.

follows:

$$
\begin{aligned}
Po_{BF_{pipelined_4}} &= Po_{hash_{h_1}} + Po_{lookup_{h_1}} \\
&+ p_{set_{h_1}} \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) \\
&+ p_{set_{h_1}} p_{set_{h_2}} \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) \\
&+ p_{set_{h_1}} p_{set_{h_2}} p_{set_{h_3}} \left(Po_{hash_{h_4}} + Po_{lookup_{h_4}}\right) + ... \\
&+ p_{set_{h_1}} p_{set_{h_2}} p_{set_{h_3}} \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) \\
&+ Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
&= Po_{hash_{h_1}} + Po_{lookup_{h_1}} \\
&+ A \left(Po_{hash_{h_2}} + Po_{lookup_{h_2}}\right) \\
&+ A^2 \left(Po_{hash_{h_3}} + Po_{lookup_{h_3}}\right) \\
&+ A^3 \left(Po_{hash_{h_4}} + Po_{lookup_{h_4}}\right) + ... \\
&+ A^3 \left(Po_{hash_{h_k}} + Po_{lookup_{h_k}}\right) \\
&+ Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
&= \sum_{i=1}^{3} A^{i-1} \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) \\
&+ \sum_{i=4}^{k} A^3 \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}}\right) \\
&+ Po_{Gate_{and_4}} + Po_{Gate_{and_{k-3}}} \\
&\quad with\ A = \left(1 - e^{\frac{-kn}{m}}\right)
\end{aligned}
\tag{4.39}
$$

In Equation (4.39), if we consider the power consumed by '$Gate_{and_4}$' and '$Gate_{and_{k-3}}$' (the power consumed by 4-input and (k-3)-input *'and'* gates), it is equal to the power consumed by '$Gate_{and_k}$' then Equation (4.39) is rewritten as follows:

$$Po_{BF_{pipelined_4}} = \left(\sum_{i=1}^{3} \left(1 - e^{\frac{-kn}{m}} \right)^{i-1} + \sum_{i=4}^{k} \left(1 - e^{\frac{-kn}{m}} \right)^{3} \right) \left(Po_{hash_{h_i}} + Po_{lookup_{h_i}} \right) + Po_{Gate_{and_k}} \tag{4.40}$$

4.4 Summary

In this chapter, we presented three approaches to make more efficient Bloom filters.

- In the first approach, we presented a new technique to embed a multi-level cache memory in a counting Bloom filter (CCBF). Using the counting Bloom filter property, the number of accesses and sizes of the l-level and 3-level cache in the CCBF architecture were investigated. To realize the analysis and simulation results, we implemented a software packet classifier in basic tuple space using a $H3$ class of universal hashing functions.

- In the second approach, we presented an innovative technique to decrease the memory usage in the standard Bloom filter. We utilized an additional hashing function to select a generated address by hashing functions in the Bloom filter. Utilization of an additional hashing function increases the performance of Bloom filter (in term of average bucket size, maximum search length and number of collisions) in comparison to the standard Bloom filter. Our analysis and software implementation results validate this. The main advantage of the BFAH technique is that it be applied to all Bloom filter types but the pruning technique only works with counting Bloom filter. The pruning technique needs a parallel search among the counters pointed to by hashing functions to find the counter with minimum value while the BFAH approach accesses the item in question by only one hash operation.

- In third approach, we presented a k-stage pipelined Bloom filter architecture to decrease the power consumption in packet classification. The

performed analysis show that the pipelined Bloom filter architecture decreases power consumption in comparison to the standard Bloom filter. Our observation of the software packet classifier for real packet traces shows that the first three stages of the pipelined Bloom filter detect most of the mismatch packets, therefore, a 4-stage pipelined Bloom filter is sufficient to classify packets. The 4-stage pipelined Bloom filter is more appropriate than standard Bloom filter when the power consumption is critical.

These approaches are useful in the design of low power and high-performance memory architectures and processing engines (e.g., forwarding, packet introspection and classification) utilized in the network processors and network processing applications.

Chapter 5

Optimal Bandwidth Allocation in Network Processing Systems

The bandwidth growth and applications variety fueled the need for high-performance network processors (NPs). The packet processing tasks have specific requirements in term of response time and throughput. The traditional NP consumes many cycles when it needs to communicate with other networking elements. Therefore, the utilization of more powerful network processors should improve the communication between NPs and boost the overall performance within the network. A valuable tool to analyze such networks is the queuing network model. The queuing network model can be utilized to derive a model for network processors for packet processing system in a grid-oriented environment. In this model, it is important to be able to determine how to best allocate the arrival rate (bandwidth) in such a manner as to optimize various performance measures, such as the response time and the number of items in the network.

In this chapter, we present a queuing model for the NP using Jackson model in Section 5.1. The optimal bandwidth allocation is presented in Section 5.2. Section 5.3 summarizes the main points in this Chapter.

5.1 NP-based Architecture Model

In this section, we present the simple abstract NP model and an NP-based architecture model. Subsequently, we present the optimal arrival rate allocation concept.

5.1.1 Simple Abstract NP Model

The handling of incoming packets by a network processor can be separated into two planes, i.e., the data plane and the control plane, that differ in speed and the manner in which packets are handled. In the data plane, simple and highly repetitive tasks are performed. Most packets pass through this high-speed plane of an NP. In the control plane, exceptional packets and complex routines are handled. This model is depicted in Figure 5.1(a).

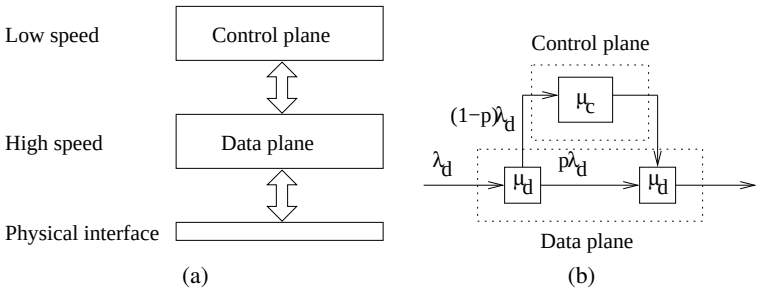


Figure 5.1: (a) Simple abstract NP model. (b) Simple abstract NP queuing model.

Based on the abstract NP model, we can derive a queuing model with the mapping of each plane on a separate queue. The related queuing model is depicted in Figure 5.1(b). We call this model the Abstract NP Queuing (ANPQ) model [14], [10]. In this figure, the λ_d and μ_d are the arrival rate and the service rate in the data plane, respectively, and λ_c and μ_c are the arrival rate and the service rate in the control plane, respectively, and λ is the arrival rate of the overall system. Using Equation (2.4), the response time T_s in the ANPQ model is:

$$\begin{aligned} T_s &= \frac{1}{\lambda}\left(\frac{\lambda_d}{\mu_d-\lambda_d}+\frac{p\lambda_d}{\mu_d-p\lambda_d}+\frac{(1-p)\lambda_d}{\mu_c-(1-p)\lambda_d}\right) \\ &= \left(\frac{1}{(\mu_d-\lambda_d)}+\frac{p}{(\mu_d-p\lambda_d)}+\frac{(1-p)}{(\mu_c-(1-p)\lambda_d)}\right) \end{aligned} \tag{5.1}$$

In this equation probability p shows the rate of items that is forwarded to data plane. In addition, in this equation each term shows the response time related to each queue in data and control plane.

5.1.2 Model Overview of Grid-oriented NP Network

In this section, the grid-oriented NP architecture model is presented. This model is not for specific NPs or their internal components such as buses and memories. This model investigates the role of NPs as processing elements to process incoming packets in a grid computing environment. In this model, one of the NPs operates as master-NP and others cooperate as slave-NPs. When the master-NP's load is saturated, it requests cooperation from other NPs that have a low load. After finding an NP as a slave, the master-NP defers part of data packets to it for processing. The functions of a master-NP include platform configuration and reconfiguration, load balancing, packet processing, scheduling, management, and accounting of the slave-NPs. In this platform, each NP can operate as a slave or a master at different times, it depends on the condition of the NPs. The master-NP segregates input packets between slave-NPs if it needs more processing power. When some packets cannot be handled by slave-NPs, these will be forwarded back to the master-NPs. The master-NP sends the packet stream to slave-NPs using direct path and receives control and non-handled packets from slave-NPs. In the modeling, the behavior of slave-NPs is evaluated based on the master-NP, therefore, the master-NP is represented using the ANPQ model with slave-NPs as simple processing elements.

Model analysis of network of NPs in Grid-oriented environment

Routers and switches are important parts of network processing and grid computing networks. These hardware resources are comprised mainly of network elements that are called network processors (NPs). The NPs in this platform are spread in different network environments and locations. The general architecture is depicted in Figure 5.2.

In this architecture, the collaborative processing is the main ambition where the one of NPs can operate as master-NP and others can collaborate as slave-NPs, the concept is depicted in Figure 5.3.

Figure 5.3 depicts different configurations to collaborate NPs together. The NP architecture in grid environments is depicted in Figure 5.4. In this figure,

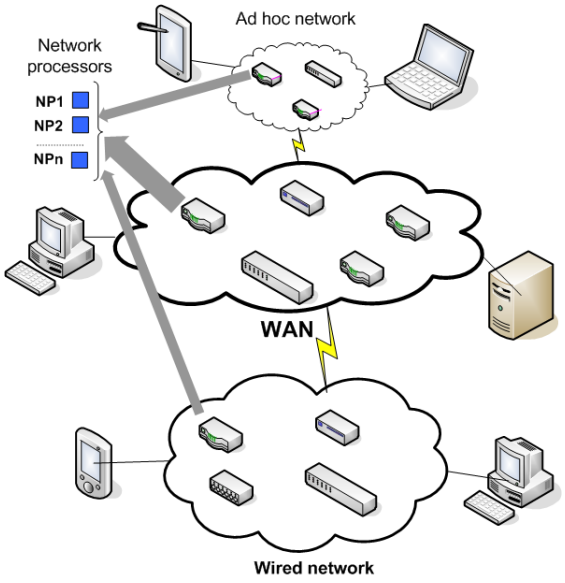


Figure 5.2: NPs distribution in grid environment.

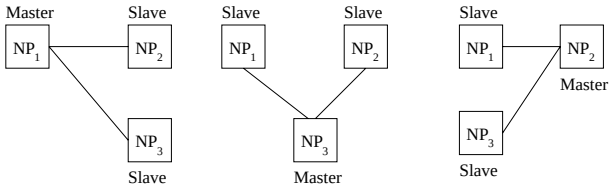


Figure 5.3: Different configuration of NPs.

NP1 receives a packet stream S_1, divides it to other streams, and sends those to slave-NPs.

The model comprises two parts: a master-NP and a set of slave-NPs. The master-NP includes two data plane processing units D and P1 and a control plane processing unit C (based on the ANPQ model). We can observe that the data plane processing unit receives the packet stream S_1 and divides it between different slave-NPs. In this figure, the S_i (with $i \geq 2$) represents external arrival rate to different slave-NPs, P_{di} (with $i \geq 1$) represents the probabilities of internal arrival rate between master-NP and slave-NPs and called the *forward routing probability*. P_{id} (with $i \geq 2$) represents the probabilities of internal

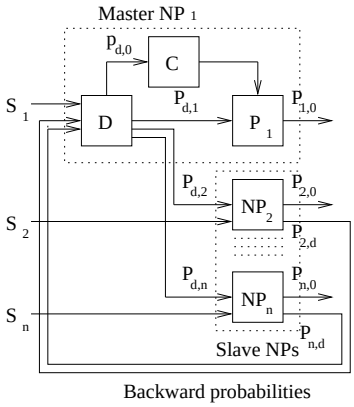


Figure 5.4: Model of a grid-oriented NP.

arrival rate among slave-NPs and master-NP and called the *backward routing probability*. The values of the backward routing probabilities is zero when the slave-NPs can handle all packet streams. P_{i0} represents the probability of the outgoing stream for each NP, λ_i and μ_i represent the arrival rate and service rate for different slave-NPs, respectively, λ_d represents the arrival rate for the data plane processing unit of master-NP, λ_c represents the arrival rate for control plane processing unit of master-NP, μ_d represents service rate for data plane processing unit, μ_c represents service rate for control plane processing unit, and μ_i represents service rate for different slave-NPs. Using Equations 2.1 and 2.4 we can write the arrival rate equations for different NPs in Figure 5.4 as following:

$$\begin{aligned} \lambda_d &= \frac{s_1 + \sum_{i=2}^{n} s_i p_{id}}{1 - \sum_{i=2}^{n} p_{di} p_{id}} \\ \lambda_1 &= p_{d1}\lambda_d \\ \lambda_i &= s_i + p_{di}\lambda_d, i = 2...n \\ \lambda_c &= p_{d0}\lambda_d \end{aligned} \tag{5.2}$$

Using the Equation (2.4), the mean response time for our model can be determined as follows:

$$\begin{aligned} T_s &= \frac{1}{\lambda}\left(\frac{\lambda_d}{\mu_d - \lambda_d} + \frac{\lambda_c}{\mu_c - \lambda_c} + \sum_{i=1}^{n}\frac{\lambda_i}{\mu_i - \lambda_i}\right) \\ &= \frac{1}{\lambda}\left(\frac{\lambda_d}{\mu_d - \lambda_d} + \frac{p_{d0}\lambda_d}{\mu_c - p_{d0}\lambda_d} + \frac{p_{d1}\lambda_d}{\mu_d - p_{d_1}\lambda_d}\right. \\ &+ \left.\sum_{i=2}^{n}\frac{s_i + p_{di}\lambda_d}{\mu_i - s_i - p_{di}\lambda_d}\right) \end{aligned} \tag{5.3}$$

In Equation (5.3), we can observe that for some values the denominator of different terms can be zero therefore, the response time T_s will be increased. Since some NPs are busy and can not handle the incoming packets. Consequently, the slave-NPs with lower load and response time should be selected. Using Equations (5.2) and (5.3), the model response time equation is extended as follows:

$$\begin{aligned} T_s &= \frac{1}{\lambda}\left(\frac{C}{\mu_d - C} + \frac{p_{d0}C}{\mu_c - p_{d0}C} + \frac{p_{d1}C}{\mu_d - p_{d_1}C}\right. \\ &+ \left.\sum_{i=2}^{n} \frac{s_i + p_{di}C}{\mu_i - s_i - p_{di}C}\right) \\ & with\ C = \frac{s_1 + \sum_{i=2}^{n} s_i p_{id}}{1 - \sum_{i=2}^{n} p_{di} p_{id}} \end{aligned} \tag{5.4}$$

In the Equation (5.4), λ is the entire system arrival rate and equal to $\sum_{i=1}^{n} s_i$.

5.2 Optimal Arrival Rate Allocation

Based on the model described in previous section, we can observe that the value of the arrival rate for each slave-NP is determined by the master-NP, but how are the optimal arrival rates determined? In other words, we should find the answers for these questions: How is the value of forward routing probability p_{di} determined? Which slave-NPs can decrease/increase the response time? If an slave-NP increases overall system response time how can this problem be overcome? Therefore, we utilize an optimal arrival allocation mechanism and find a sequence of slave-NPs to minimize system response time. Afterwards, we can use the proportional allocation to distribute the incoming items between different slave-NPs. An alternative to find the optimal arrival rate in the system is the utilization of optimal capacity allocation policy that is presented in Equation (2.5). To derive the Equation (2.5), we assumed that the value of arrival rates are constant and the optimal service rate has been evaluated. Thereby, we derive a new formula to estimate the optimal arrival rates. We assume that we have control over the arrival rate λ_1, λ_2, ..., λ_M where λ_i is the arrival rate for different slave-NPs. The slave-NPs are managed and controlled by the master-NP, but with a constraint that fixes the total arrival capability to a constant value c (due to the standard communication line bandwidth) as follows: $\sum_{i=1}^{M} \lambda_i = c$. For a given set of service rates μ_i, we want to find the optimal set λ_i that minimizes the items $\bar{N} = \sum_{i=1}^{M} N_i$, where $\bar{N}$ represents the mean number of items or queue length that can be computed using Equation (2.3).

Therefore, we can derive the following equation:

$$\bar{N} = \sum_{i=1}^{M} \frac{\lambda_i}{\mu_i - \lambda_i} \; with \; constraint \; \sum_{i=1}^{M} \lambda_i = c \tag{5.5}$$

In this equation the constraint is defined based on the arrival rates λ_i. This equation is used to estimate the value of arrival rates. An alternative to minimizing Equation (5.5) is to use of Lagrangian multiplier. In mathematical optimization problems, Lagrange multipliers is a method to find the local extremum of a function of several variables subject to one or more constraints. This method reduces a problem in n variables with k constraints to a solvable problem in $n + k$ variables with no constraints [61]. Using the method of Lagrangian multipliers, Equation (5.5) is rewritten as follows:

$$H = \sum_{i=1}^{M} \frac{\lambda_i}{\mu_i - \lambda_i} + x(\sum_{i=1}^{M} \lambda_i - c) \tag{5.6}$$

To minimize H, we differentiate and obtain the following equation:

$$\frac{\partial H}{\partial \lambda_i} = \sum_{i=1}^{M} \frac{\mu_i}{(\mu_i - \lambda_i)^2} - Mx \tag{5.7}$$

If we set the derivative to zero then we find that H is minimized by $\lambda_i = \mu_i - \sqrt{\frac{\mu_i}{x}}$ substituting this expression for λ_i into $\sum_{i=1}^{M} \lambda_i = c$, we find that $\frac{1}{\sqrt{x}} = \frac{\sum_{i=1}^{M} \mu_i - c}{\sum_{i=1}^{M} \sqrt{\mu_i}}$. Hence the optimal value for the arrival rate obtained by substituting x into the optimal value for λ_i, we have:

$$\lambda_i = \mu_i - \sqrt{\mu_i}(\frac{\sum_{j=1}^{M} \mu_j - c}{\sum_{j=1}^{M} \sqrt{\mu_j}}) \tag{5.8}$$

The description of this concept is depicted in Figure 5.5.

In this figure, the curves A and B represent service and arrival rates of different NPs, respectively. Curve C represents the optimal arrival rate for these NPs, where they are estimated using Equation (5.8). We can observe that the mapping of curves B and C in the same space generates different areas called overload and underload areas containing NPs. The overload area represents saturated NPs. This means for the NPs in the overload area the arrival rate values are more than optimal arrival rates that these NPs increase the response time of

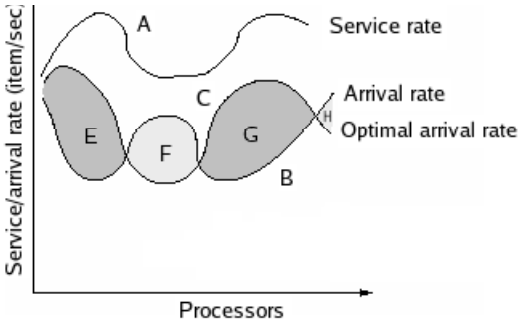


Figure 5.5: Typical curves in optimal arrival rate allocation.

whole system. The underload area represents NPs that can receive more arrival rate. In the underload area, the arrival rate values are lower than the optimal arrival rates. Based on the Figure 5.5, the areas E and G represent the underload area and areas F and H represent overload areas. The value of the arrival rate in the underload areas can increase to the optimal arrival rate values.

5.3 Summary

In this chapter, we proposed an abstract model for network processor using queueing networks (ANPQ) and open queues. Based on the ANPQ, we described the NP-based architecture model in a grid-oriented network environment using the Jackson model. In network processing environments, an important factor is to minimize the response time. Therefore, we presented an approach to optimize the rate arrival allocation. In our approach, we derive a formula that proposed a solution to select a set of NPs for packet processing, in order to minimize the total system response time. This solution can be implemented in grid-oriented environments to allocate optimal bandwidth.

Chapter 6

Performance Evaluation and Experimental Results

This chapter presents the results of the proposed approaches that were discussed in chapters 3, 4 and 5. These approaches include the performance evaluation of collaboration of reconfigurable processors in grid environments for multimedia kernels, the performance evaluation to optimize Bloom filters in network processing application and an approach for the optimal bandwidth allocation in the packet processing systems.

Section 6.1 presents the experimental results of collaboration of reconfigurable processors in grid environments for multimedia kernels. Section 6.2 presents the simulation and implementation results of three approaches in Bloom filters. The simulation results of approach for optimal bandwidth allocation for network processors is described in Section 6.3. Finally, Section 6.4 summarizes this chapter.

6.1 Collaborative Reconfigurable Processors in Grid Environments

In this section, we present the experimental results which have been obtained using the CRGridSim simulator [15], [115]. First, we present the application mapping on CRGC for multimedia kernels. Subsequently, the performance evacuation of CRGC is presented.

6.1.1 Application Mapping on CRGC

Parallel architectures can be programmed using two models of parallel programming, data parallelism and task parallelism. In data parallelism, data is partitioned and distributed among the processing elements. For example, in case of image processing applications, images are split into several sub-images and each sub-image is processed by a processing element. In task parallelism, the instructions can be grouped into tasks and each task is assigned to a processing element. In other words, the task is split into a number of subtasks and each subtask is assigned to a specific processor. In addition, the data necessary for each subtask is sent to the appropriate processing element.

Based on the simulation environment presented in Chapter 3.4, we map our applications on two different configurations. The first configuration is the collaboration of GPPs with packet size of 64 KBytes. The second configuration is the collaboration of reconfigurable processor (elements) with packet size of 64 KBytes. Tables 6.1 and 6.2 show the mapping of the 2D DWT on the first and second configurations, respectively. GPP0 is the main processing element, while other processors are the collaborator processing elements. Columns three to six represent the number of assigned gridlets from each group (Table 3.2) to each processing element. For example, in Table 6.1, GPP1 processes 3, 0, 2, and 2 gridlets from groups 1, 2, 3, and 4, respectively. The seventh column shows the total number of processed gridlets by each processing element. The last column represents the total number of executed instructions by each processing element. For instance, in Table 6.1, GPP1 executes 3×35, 0×46, 2×84, and 2×267 MIs for groups 1, 2, 3, and 4, respectively. The total executed instructions (see Table 3.2) are 807 MIs (million of instructions).

Three collaborator processing elements in Table 6.1 process $7 + 8 + 10 = 25$ gridlets, while the main processing element processes 15 gridlets. It should be noted that these numbers are obtained by the executing of simulation program presented in Chapter 3.4. In other words, the GPP0 processes the most number

Resource		Gridlets (based on table 3.2)					Total # of
Type	MIPS	# of assigned gridlets from Group 1	# of assigned gridlets from Group 2	# of assigned gridlets from Group 3	# of assigned gridlets from Group 4	Total # of assigned gridlets	ins. for PEs
GPP0	30	2	5	4	4	2+5+4+4=15	1704
GPP1	35	3	0	2	2	3+0+2+2=7	807
GPP2	50	2	2	1	3	2+2+1+3=8	1047
GPP3	40	3	3	3	1	3+3+3+1=10	762

Table 6.1: Application mapping of the 2D DWT on collaboration of GPPs on a grid computing.

Resource		Gridlets (based on table 3.2)					Total # of
Type	MIPS	# of assigned gridlets from Group 1	# of assigned gridlets from Group 2	# of assigned gridlets from Group 3	# of assigned gridlets from Group 4	Total # of assigned gridlets	ins. for PEs
GPP0	30	2	2	2	3	2+2+2+3=9	1131
RE1	35	3	3	3	1	1+3+3+3=10	762
RE2	50	1	3	3	3	1+3+3+3=10	1226
RE3	40	4	2	2	3	4+2+2+3=11	1201

Table 6.2: Application mapping of the 2D DWT on collaboration of reconfigurable processors (elements) on a grid computing.

of gridlets. On the other hand, three collaboration reconfigurable processors in Table 6.2 process $10 + 10 + 11 = 31$ gridlets, while the main processing element processes 9 gridlets, which is the least number of gridlets compared to other processing elements. As a result, the collaboration of reconfigurable processors can process more gridlets than the collaboration of GPPs.

In order to increase the computational time, both media kernels are integrated and executed together. Table 6.3 depicts the mapping of the execution of both media kernels on the collaboration of GPPs, while Table 6.4 depicts the mapping of the execution of both media kernels on the collaboration of reconfigurable processors. Three collaborator processing elements in Table 6.3 process $8+13+10 = 31$ gridlets, while the main processing element processes 9 gridlets. On the other hand, three collaboration reconfigurable processors in Table 6.4 process $12+13+12 = 37$ gridlets, while the main processing element processes 3 gridlets. As a result, in collaboration mode, each collaborator processing element processes much more gridlets than the main GPP.

6.1.2 Performance Evaluation

In order to evaluate the proposed approach, we considered two packet sizes, 32 KBytes and 64 KBytes (the largest packet sizes in the networks). Our results show that using larger packet sizes lead to higher performance than smaller

Resource		Gridlets (based on table 3.2)					Total # of ins. for PEs
Type	MIPS	# of assigned gridlets from Group 1	# of assigned gridlets from Group 2	# of assigned gridlets from Group 3	# of assigned gridlets from Group 4	Total # of assigned gridlets	
GPP0	30	2	3	2	2	2+3+2+2=9	2588
GPP1	35	2	2	1	3	2+2+1+3=8	2978
GPP2	50	2	4	5	2	2+4+5+2=13	3438
GPP3	40	4	1	2	3	4+1+2+3=10	3290

Table 6.3: Application mapping of the 2D DWT+co-occurrence matrix on collaboration of GPPs on a grid computing.

Resource		Gridlets (based on table 3.2)					Total # of ins. for PEs
Type	MIPS	# of assigned gridlets from Group 1	# of assigned gridlets from Group 2	# of assigned gridlets from Group 3	# of assigned gridlets from Group 4	Total # of assigned gridlets	
GPP0	30	0	0	1	2	0+0+1+2=3	1760
RE1	35	4	4	2	2	4+4+2+2=12	2916
RE2	50	3	3	3	4	3+3+3+4=13	4447
RE3	40	3	3	4	2	3+3+4+2=12	3167

Table 6.4: Application mapping of the 2D DWT+co-occurrence matrix on collaboration of reconfigurable processors (elements) on a grid computing.

packet sizes. Larger packet sizes decreases the communication overhead due to sending less packets. In our case, we have 40 images with a total size of 294 MBytes. These images have 4539 packets of 64 KBytes and 9219 packets of 32 KBytes. In addition, we considered four different configurations, collaboration of 3 GPPs, a GPP with 2 REs, 4 GPPs, and a GPP with 3 REs. The configuration of 3 GPPs means that 2 GPPs are collaborating with a main GPP. The star topology has been used for the collaboration mechanism that a structure of this topology was depicted in Figure 3.2 (a).

Figure 6.1 depicts the speedups of the first two configurations, 3 GPPs and a GPP with 2 REs over a GPP for different packet size. Figure 6.2 also depicts the speedups of the last two configurations, 4 GPPs and a GPP with 3 REs, over a GPP for different packet size. Our observations from these figures are the following. First, increasing the packet size from 32 KBytes to 64 KBytes improves the performance. As can be seen that the speedups for the packet size 64KB are larger than the speedups for packet size 32KB. This is due to the fact that larger packet sizes decreases the communication overhead. Second, the collaboration of reconfigurable processors improve the performance more than the collaboration of GPPs. For our configuration, the performance improvement of the collaboration of reconfigurable processors over the collaboration of GPPs is of up to 2.5. This is because, based on the specifications of the processing elements in the simulation environment in Tables 3.3 and 3.4

each reconfigurable processor processes more instructions than a GPP. Finally, executing computational intensive applications yields much more performance than non-computational intensive applications.

This is because the impact of the communication overhead will be reduced compared to the computational time. As it can be seen in those figures, combination of both kernels increases the computational time and then it obtains more speedups than the execution of each kernel separately. Additionally, increasing the number of collaborator processing elements improves the performance. This is because the submitted subtasks to each collaborator are decreased. This reduces the number of processed instructions by each processing element. Furthermore, from Figures 6.2 (a) and (b) it can be observed when the processing power of RE changes from 2 to 5 the speedup of the system is increased.

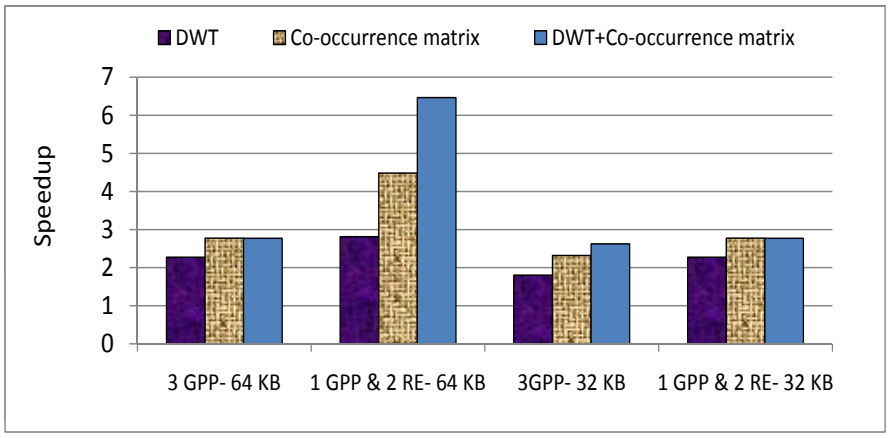
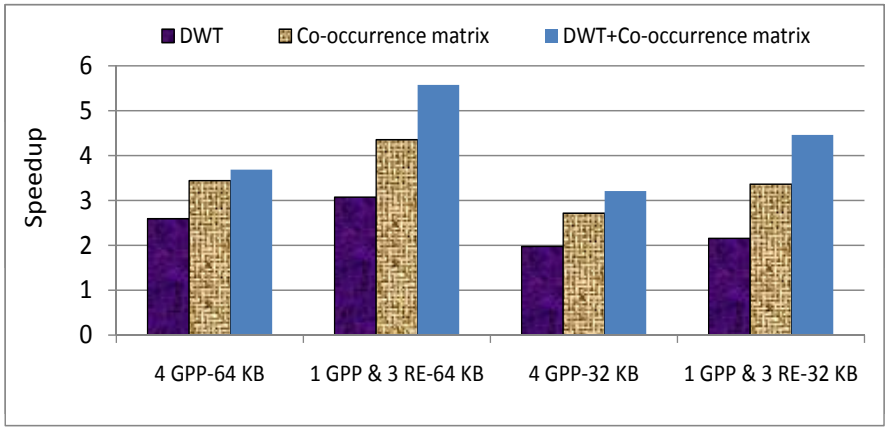
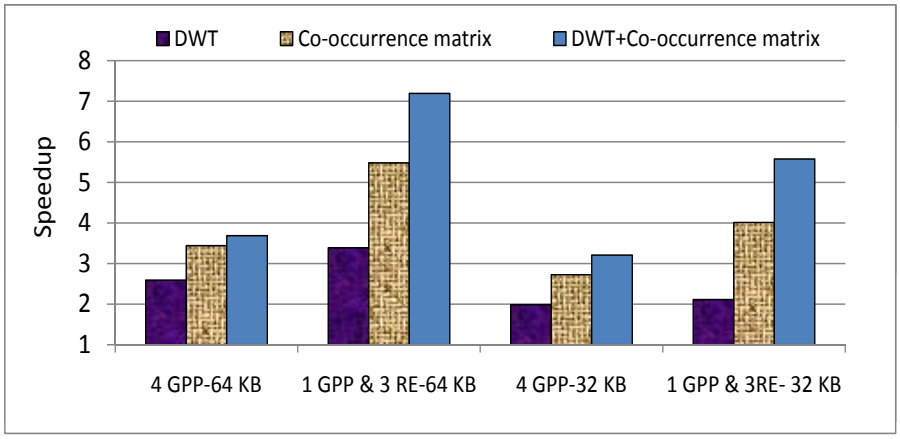


Figure 6.1: Speedup for different configurations with 2 collaborator processing elements over one GPP.

It should be noted that the theoretical upper bound of speedup that can be obtained in CRGC approach is 15x using Equation (3.10), while the actual speedup is 7.2x. The reason for this difference in the speedup is due to the impact of following factors, communication time, reconfiguration time, application mapping policy, and load balancing and application structure have been considered. In addition, in a real network, each processing element can collaborate with a limited number of neighbor processing elements. Increasing the number of neighbor processing elements increases the communication time and this reduces the performance.



(a)



(b)

Figure 6.2: Speedup for different configurations with 3 collaborator processing elements over one GPP. (a) When reconfigurable elements (REs) are 2 times faster than GPP. (b) When reconfigurable elements (REs) are 5 times faster than GPP.

6.2 Bloom Filter Architectures Results

In this section, the specification of different rule-set databases for the testing purpose is introduced. Subsequently, the experimental results including simulation and implementation results for different Bloom filter approaches are presented.

6.2.1 System Testing

For the testing purposes, we utilize different rule-set databases and packet traces that have been used by the Applied Research Laboratory in Washington University in St. Louis [94]. The specification of the rule-set databases and packet traces is presented in Table 6.5.

Rule Database	**FW1-100**	**FW1-1k**	**FW1-5k**	**FW1-10k**	**FW1**	**IPC1**	**ACL1**
Number of rules	92	971	4653	9311	266	1550	752
Number of tuples	26	42	52	57	36	179	44
Packet trace	**FW1-100**	**FW1-1k**	**FW1-5k**	**FW1-10k**	**FW1**	**IPC1**	**ACL1**
Number of Packets	920	8050	46700	93250	2830	17020	8140

Table 6.5: Rule-set database and packet trace specification.

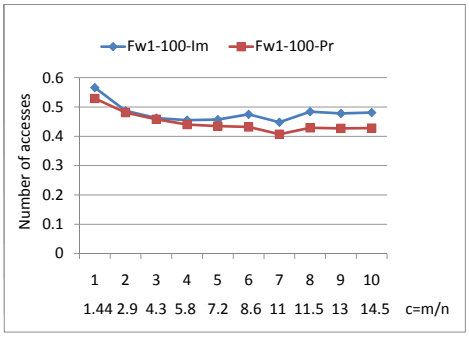
Table 6.5 includes seven rule-set databases and correspondent packet traces. The rule-sets FW1, ACL1, and IPC1 are extracted from real rule-sets and others generated by the Classbench benchmark. More details on Classbench, rule-set databases, and packet traces can be found in [94].

In the rule-set database, each rule consists of 5 header fields including "[Source IP address, Destination IP address, Source port, Destination port, Protocol]" and the format is *"@[Source IP address prefix in dot-decimal notation]/[Prefix length] [Destination IP address prefix in dot-decimal notation]/[Prefix length] [Low source port] : [High source port] [Low destination port] : [High destination port] [Protocol value in hexadecimal]/[Protocol mask in hexadecimal]"*. An example of a rule in the rule-set database is : *"@204.152.188.80/28 204.152.188.64/28 67 : 67 67 : 67 0x11/0xff"*. The packet header trace format is *"[Source IP address in decimal] [Destination IP address in decimal] [Source port value in decimal] [Destination port value in decimal] [Protocol in decimal]"*. An example of a packet header in a packet trace is: *"3337533518 2390673931 65535 65535 1 9"* [6], [7], [94].

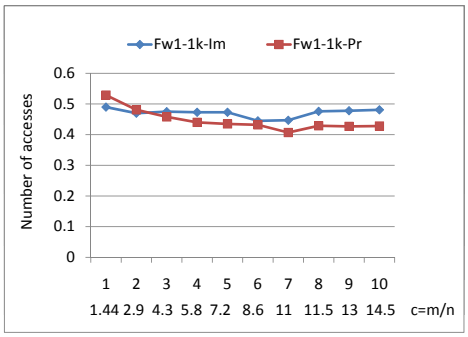
6.2.2 Cache Counting Bloom Filter

In this subsection, we present the implementation results of the CCBF architecture in packet classification using tuple space search and subsequently compare them to the simulation results. The implementation and simulation results for Fw1-100, Fw1-1k and Fw1-5k rule-set databases are depicted in Figure 6.3.

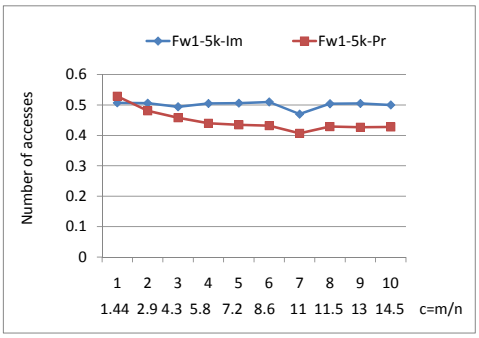
In this figure, 'Fw1-xx-Im' shows the graph of the software implementation and 'Fw-xx-Pr' shows the graph of mathematical simulation results that are predicted



(a)



(b)



(c)

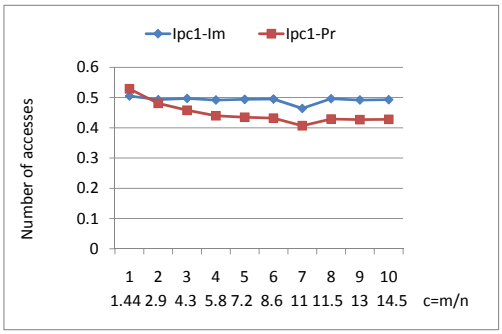
Figure 6.3: The number of accesses in CCBF normalized to the number of accesses in standard Bloom filter include mathematical simulation and software implementation. (a) The number of accesses in a 3-level CCBF for Fw1-100. (b) The number of accesses in a 3-level CCBF for Fw1-1k. (c) The number of accesses in a 3-level CCBF for Fw5-1k.

from Equation (4.12). The vertical axis shows the number of accesses that is normalized to nk (n is number of items and k is number of hashing functions). The horizontal axis includes two sequences with the first sequence specified by k (number of hashing functions) and the second sequence is specified by $c = m/n$ (m represents the size of address space in counter-array in CCBF and n represents the number of items). The value of $c = m/n$ is determined using Equation (2.6) that estimates the optimal value for k to have the minimum false positive probability. These rule-set databases (Fw1-100, Fw1-1k and Fw1-5k) are synthetic that were generated by the Classbench benchmark. The software implementation and mathematical simulation results for Fw1, Ipc1 and average of all rule-set databases are depicted in Figure 6.4.

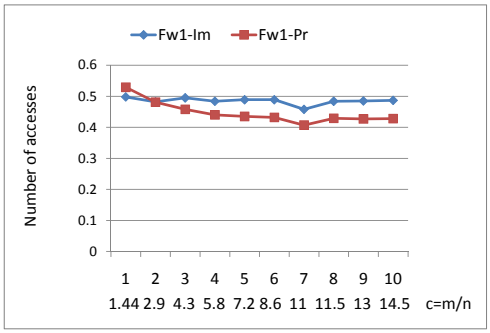
Figs. 6.4 (a) and 6.4 (b) depict the number of accesses for Fw1 and Ipc1 rule-set databases that were extracted from real rule-set databases. Figure 6.4 (c) depicts the average for all of utilized rule-set databases.

As we expected the mathematical simulation results verify the software implementation results although in the software implementation results number of accesses are more than mathematical simulation results. The maximum difference between implementation and simulation results is 7.8%. This difference is due to the following facts: number of tuples, distribution of rules inside the tuples, and utilized hashing functions. In the packet classification using tuple space, the number of tuples and the number of rules in the tuples are constant for different rule-set databases and different tuples in each rule-set database. In most rule-set databases, one tuple includes about half of the rules and some tuples only have one or several rules. In the simulation analysis, the simulation results were obtained by investigating a CCBF with a big bit-array and a single set of items. The $H3$ hashing function selection procedure is random causing some of them to generate more collisions that needs more accesses. If we can utilize a process to select optimal $H3$ hashing functions, the number of accesses should decrease and converge to the simulation results. The total size of cache levels for real rule-set databases in a 3-level CCBF is depicted in Figure 6.5.

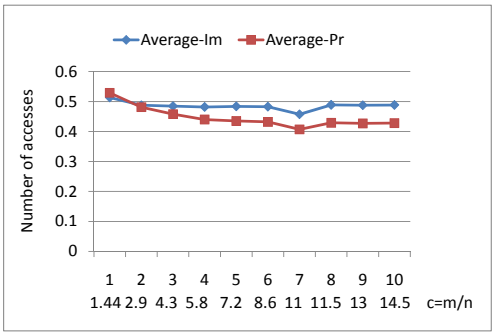
In this figure, 'rule-set-im' shows the total size of cache for rule-set database 'rule-set' where the results are extracted by a software packet classifier and normalized to nk (number of items multiply by number of hashing functions). Based on the software implementation the total cache size has some fluctuations. This is due to internal gaps of buckets in the third cache level.



(a)



(b)



(c)

Figure 6.4: The number of accesses in CCBF normalized to the number of accesses in standard Bloom filter include mathematical simulation and software implementation. (a) The number of accesses in a 3-level CCBF for Fw1. (b) The number of accesses in a 3-level CCBF for Ipc1. (c) The average number of accesses in a 3-level CCBF for all of utilized rule-set databases.

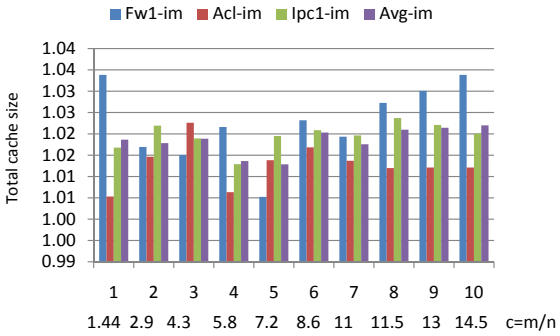


Figure 6.5: The total size of cache in CCBF normalized to the number of size of memory in standard Bloom filter.

Discussion

Our analysis and implementation show that the CCBF can be utilized for network processing. The incoming items are stored in the memory similar to traditional replacement algorithm that is called least frequently used replacement algorithms (LFU)[105]. In the CCBF, a bucket with larger counter has more reference, therefore it resides in a higher cache level with lower access time and the bucket with lower counter resides in a lower cache level. The utilization of CCBF needs some overheads to design CCBF unit in comparison to a standard Bloom filter such as: a mechanism to manage different cache levels and different counters that are implemented by a simple decoder to decode the incoming key and generated address in to related cache level. A mechanism to segment the large buckets (the buckets pointed by the counters that their size are larger than 3), since in third level normal bucket size is set to 3 therefore, larger counter should be segmented in to different buckets. To organize these buckets we need to utilize some buffering techniques. From the Figure 6.5, we can observe some difference between the software implementation and simulation results in term of cache size. Therefore, in hardware implementation this problem must be overcome. To eliminate this issue, we utilize the following mechanisms:

- Shared global overflow area
- Level overflow area

A shared global overflow area is a memory space where overflow items are stored. When the incoming item can not be stored on a level it is stored to the shared global overflow area. The second solution is a level overflow area that is allocated an additional memory for each level. This solution is more practical to implement since the size of each level should be assumed larger than size of level in simulation results.

6.2.3 Memory optimized Bloom Filter Using an Additional Hashing Function

In this section, we present the implementation results of a software packet classifier that utilizes the standard, pruned counting Bloom filters and a BFAH in tuple space packet classification and consequently show the results. In the figures that will be presented in next subsections the horizontal axis shows two sequences of data. The sequence with label k shows the number of hashing functions and later one shows the corresponding value of m/n that is calculated based on Equation (2.6) to minimize false positive probability. In addition, the graphs show the average bucket size, maximum search length and number of collisions for all of the rule-set databases in Table 6.5.

Investigation average bucket size

In Section 4.2.2, we analyzed the average bucket size and observed that the average bucket size in the BFAH is shorter than standard Bloom filter. We present the average bucket size for the standard Bloom filter, pruned counting Bloom filter and BFAH using a software packet classifier. The graph of average bucket size for standard Bloom filter, pruned counting Bloom filter and BFAH is depicted in Figure 6.6.

In Figure 6.6, the vertical axis shows the average bucket size in terms of the items per bucket. From this figure, we can observe that the BFAH has the average bucket size shorter than the standard Bloom filter and longer than the pruned counting Bloom filter. Additionally, Figure 6.6 depicts that the average bucket size of BFAH and pruned counting Bloom filter is very close. For instance, when $m/n = 26$ and $k = 18$, the difference is 2%. Furthermore, the increase in the number of hashing functions and the corresponding m/n value decrease the average bucket size in BFAH and pruned counting Bloom filters. It should be noted, when $k = 1$, three mentioned Bloom filters operate as a simple hashing system.

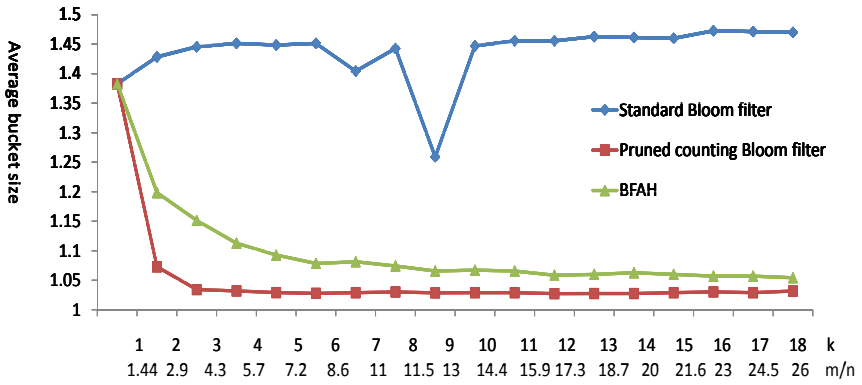


Figure 6.6: Average bucket size for the standard Bloom filter, pruned counting Bloom filter and BFAH.

Investigation maximum search length

In Section 4.2.2, we analyzed the maximum search length for standard Bloom filter, pruned counting Bloom filter and BFAH. In this section, we present the maximum search length of the standard Bloom filter, pruned counting Bloom filter and BFAH using a software packet classifier. The graph of maximum search length for standard, BFAH and pruned counting Bloom filters is depicted in Figure 6.7.

From Figure 6.7, we can observe that the maximum search length for BFAH, and pruned counting Bloom filter is almost the same. In BFAH, the maximum search length is changed between 3 and 4 while for the pruned counting Bloom filter the maximum search length is 3. We can observe that when the m/n is increased maximum search length for BFAH and pruned counting Bloom filter converge to the same value. In the standard Bloom filter, increasing the number of hashing functions increases the maximum search length. This is due to the following reasons: first, the hashing functions are not uniform, and second the generated addresses by different hashing functions for each incoming item are mapped to the same address. While in the BFAH and pruned counting Bloom filter only one out of k address is selected.

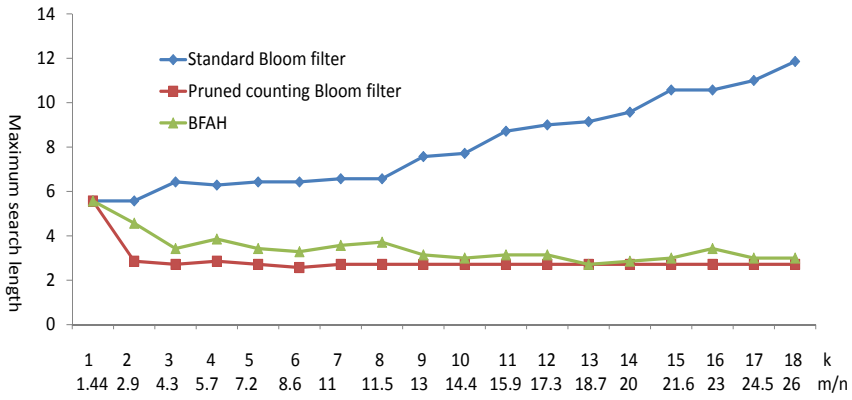


Figure 6.7: Maximum search length for standard, BFAH and pruned counting Bloom filters.

Investigation number of collisions

The graph of the number of collisions for standard, BFAH, and pruned counting Bloom filters is depicted in Figure 6.8. In this figure, the average number of collisions for all rule-set databases is normalized to n (number of rules in rule-set database) for the pruned counting Bloom filter and BFAH. For the standard Bloom filter, the number of collisions is normalized to nk. This is because, in the BFAH and pruned counting Bloom filters n rule insertions and in the standard Bloom filter nk rule insertions are performed.

Based on Figure 6.8, we can observe that the number of collisions for standard and pruned counting Bloom filters remain at a constant level, and the number of collisions for BFAH converges to the number of collisions in the pruned counting Bloom filter when the value of m/n is increased.

Discussion

In general, we observed that the utilization of BFAH decreases the average bucket size, maximum search length and the number of collisions in comparison to the standard Bloom filter and is almost same in comparison with the pruned counting Bloom filter. As we expected, the analytical results are verified by the software packet classifier results. In addition, in the pruned counting Bloom

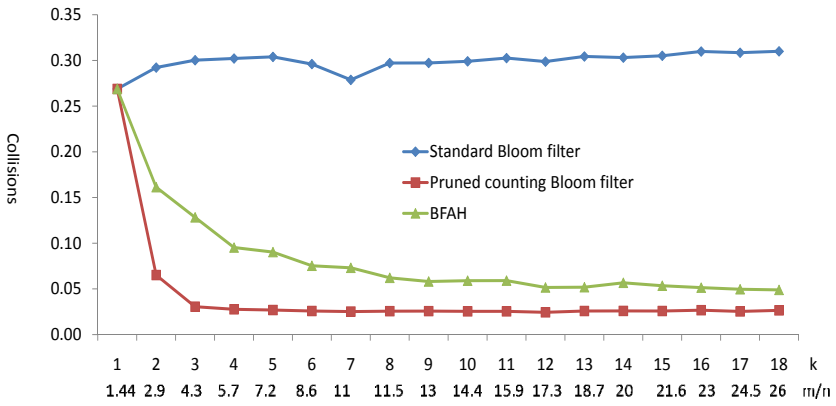


Figure 6.8: Average number of collisions for all rule-set databases that normalized to n (number of rules in rule-set database) for pruned counting Bloom filter and BFAH and normalized to nk (number of rules multiply by number of hashing functions) for standard Bloom filter.

filter, for each incoming item, k counters should be investigated but in the BFAH the address is directly selected. Furthermore, BFAH can be applied to all Bloom filters types while the pruning only works with counting Bloom filter.

The presented results show that the BFAH approach enhances different performance metrics compared to the standard Bloom filters which this means the memory bottleneck in high-performance network processing applications can be overcome.

6.2.4 k-stage Pipeline Bloom Filter Architecture Results

In this section, we present the simulation and experimental results of our k-stage pipelined Bloom filter architecture. The simulation results are based on a mathematical analysis of our architecture using Maple v.12.0 and the implementation results were generated using a software packet classifier[10].

k-stage pipeline Bloom filter results

The coefficient rate for configurations $k = ln(2)m/n$ that generates a minimum false positive probability is depicted in Figure 6.9.

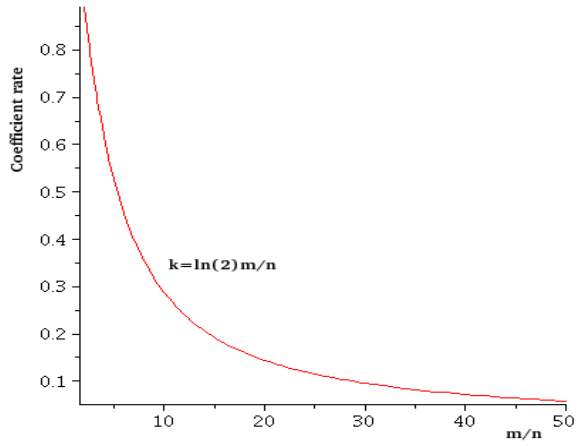


Figure 6.9: Coefficient rate in k-stage pipelined Bloom filter for configuration $k = ln(2)m/n$.

From Figure 6.9, we can observe that the growth of the number of hashing functions increases the power saving rate. The simulation results show that the utilization of pipelining in the Bloom filter decreases power consumption in comparison to a standard Bloom filter. To realize the idea, we implemented a software packet classifier and analyzed real packet traces to examine the capability of the proposed solution.

We utilize a software packet classifier and determine the average number of '0's in the bit-array of the Bloom filter that belonged to the biggest tuple. Our observations in the execution of the software packet classifier show that when the tuples are created the biggest tuple contains more than half of the rules. It should be noted that the software packet classifier was tested and results were generated for different packet traces and their correspondent rule-set databases. The average number of '0's pointed to by hashing functions in the bit-array for different packet traces is depicted in Figure 6.10.

In Figure 6.10, three configurations are represented. In all these configurations, the size of bit-array to have a minimum false positive probability is computed from Equation (2.6). Since for the given number of hashing functions and the number of rules, the size of bit-array in Bloom filter is computed. *'configuration 1'* represents a Bloom filter with 8 hashing functions and bit-array size $m = 11.2n$ (n is the number of rules). *'configuration 2'* represents a Bloom filter with 15 hashing functions and bit-array size $m = 21.6n$ and *'configuration 3'* represents a Bloom filter with 4 hashing functions and bit-array size $m =$

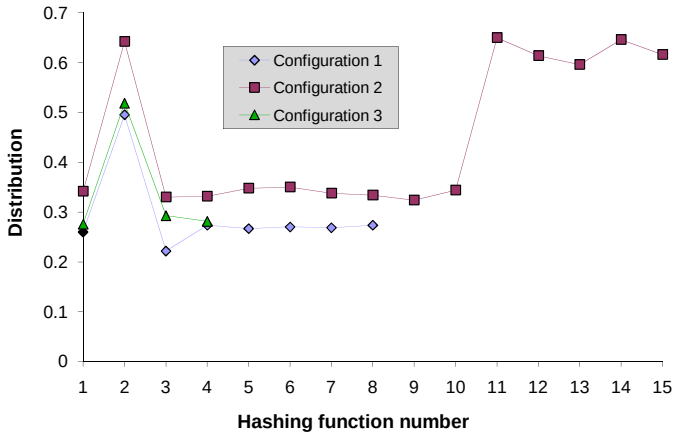


Figure 6.10: Average number of '0's in the bit-array of Bloom filter of biggest tuple using software packet classifier for three different Bloom filter configurations in the membership checking stage ('*configuration 1*' with 8 hashing functions, *'configuration 2'* with 15 hashing functions and *'configuration 3*' with 4 hashing functions).

$5.1n$. Based on the graphs in Figure 6.10, we can observe that for different configurations there are some zeros in the bit-array of Bloom filter that are pointed to by hashing functions in the membership checking stage. Since after the bit-array creation in the Bloom filter programming stage using rule-set databases, we check the different bits in the bit-array pointed to by hashing functions in the membership checking stage. After the utilization of all packet traces for each configuration, we normalize the number of zeros for each packet trace to the number of packets in a packet trace and compute the average number of zeros in all packet traces. Therefore, we conclude that in membership checking stage of Bloom filter there are some zero bits in the bit-array that are pointed to by hashing functions. We can decrease the power consumption in the Bloom filter when it is utilized in packet classification by controlling hashing functions. A question that should be addressed is related to designing of the pipelined Bloom filter architecture. How can the number of zero's be utilized in the design of the pipeline architecture? We investigate the number of mismatched packets in the different bits (pipeline stages) in the bit-array of Bloom filter in the membership checking stage. The mismatched packet detection rate is depicted in Figure 6.11.

From Figure 6.11, we can observe that more than 75% of mismatched packets

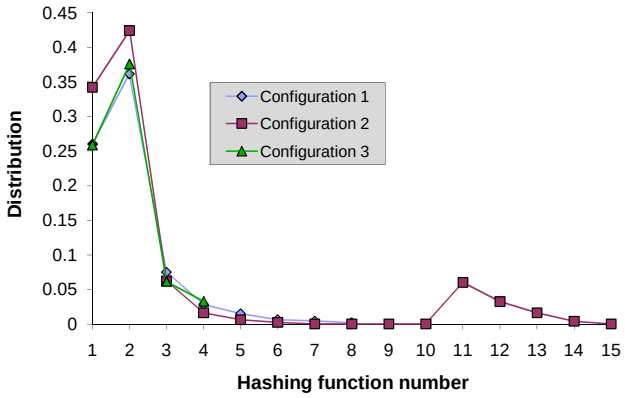


Figure 6.11: Average mismatched packets rate for all packet traces in the biggest tuple using software packet classifier for three different Bloom filter configurations.

are detected by the first three stages and 17% of packets are matched against of rules and 8% reminder are mismatched that are detected by other pipeline stages. In the second configuration, we can observe that when the hashing function number is changed between 11 and 13 some mismatched packets are detected. To overcome the problem, we can utilize an additional pipeline stage in the architecture to include other hashing functions which detects the rest of mismatched packets. In other words, a 4-stage pipelined Bloom filter architecture is useful for packet classification where the first three stages includes first three hashing functions and last stage includes last $k - 3$ hashing functions.

4-stage pipeline Bloom filter results

From Equations (4.33), (4.37), and (4.40), which represent the power consumed for the standard, k-stage, and 4-stage Bloom filters, respectively, we can observe that the difference between them is related to their coefficients. The graph of coefficients of the k-stage pipelined Bloom filter and 4-stage pipelined Bloom filter that are normalized to the coefficients of a standard Bloom filter is depicted in Figure 6.12.

From Figure 6.12, we can observe that the coefficient rate of power consumed of the 4-stage pipelined Bloom filter is more than the k-stage pipelined Bloom filter but less than the standard Bloom filter.

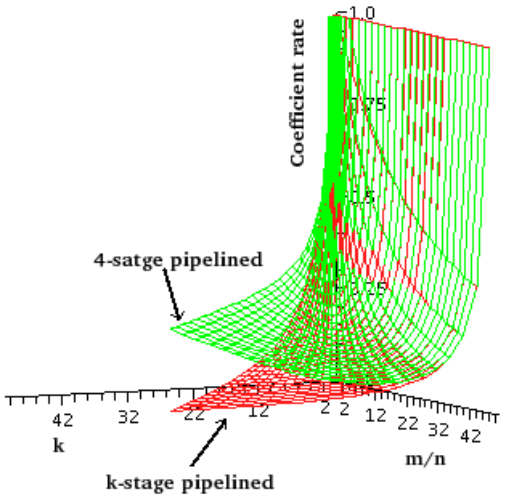


Figure 6.12: Coefficient rate in k-stage pipelined Bloom filter and 4-stage pipelined Bloom filter for different configurations in terms of k and m/n ($k \leq m/n$) that are normalized to the coefficients of a standard Bloom filter.

6.3 Network Processor Modeling

In this section, we present the simulation results of the proposed approach in Section 5. The simulation results have been generated using Maple v.10.0 Based on the general queuing model, the following assumptions have been made to derive a model in grid-oriented environment:

1. Each NP is analyzed by (M/M/1) or (M/M/c) queuing model where the incoming packets obeys the Poisson distribution. Additionally, the service time distribution is exponential.

2. In many cases, the average service rate is much greater than the average arrival rate, in this case the waiting queue would not grow too long. If the input buffer is reasonably large dropping of packets is not an issue.

In this investigation, we assume a large pool of available NPs (64 in this experiment) each having a random service and arrival rate to 'mimic' real situation as if they were are already in operation. Furthermore, out of this pool, the master-NP is allowed to choose up to 32 NPs as slave-NPs to assist itself in the processing of incoming packets. This investigation is not intended to

provide the realistic for current-day processors, but instead we are trying to determine that, out of two selection mechanisms which one is able to improve the minimum response time. The model is used as a vessel to achieve this determination as we expect its importance will become evident when more processor cores will fit on future chips. The first selection mechanism is the first-in and first-out (FIFO) mechanism that lists all possible slave NPs and deciding which one to use is solely based on which NP was entered first in the list. This is a simple selection mechanism that is easy to implement, but can have some adverse effect on the minimum response time. The second selection mechanism is optimal arrival allocation mechanism that chooses a sequence of slave-NPs to assist the master NP. This mechanism is more complex as it requires almost continuous monitoring, but it is expected to yield better results. The FIFO mechanism behavior is depicted in Figure 6.13.

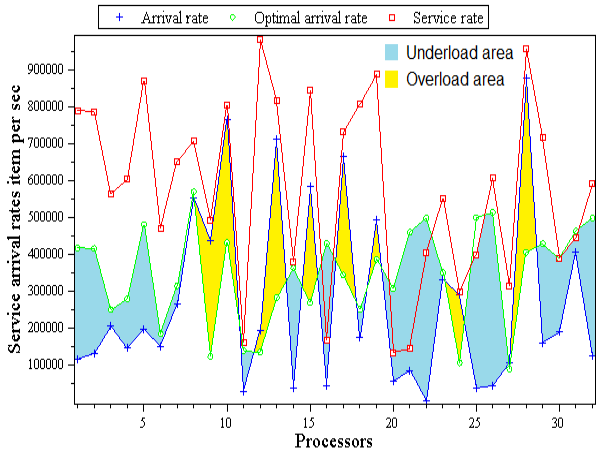
Figure 6.13 (a), depicts the service and arrival rates for different NPs based on the FIFO mechanism. The response time for NP-based architecture model in a grid-oriented environment is depicted in Figures 6.13 (b). From this figure, we can observe that, the effect of overloaded NPs in response time, since when the number of NPs is increased up to 32, the response time is not better than when the number of NP is 1. This is because of some NPs have small service rate when the 32 NPs are selected.

Therefore, the overload areas from Figure 6.13 (a) are omitted using optimal arrival rate allocation. The result is depicted in Figure 6.14 (a). The related response time for NP-based architecture in grid-oriented environment using optimal arrival allocation is depicted in Figure 6.14 (b). From this figure, we can observe that the NPs in underload areas the arrival rates are lower than optimal arrival rates resulting in better response time.

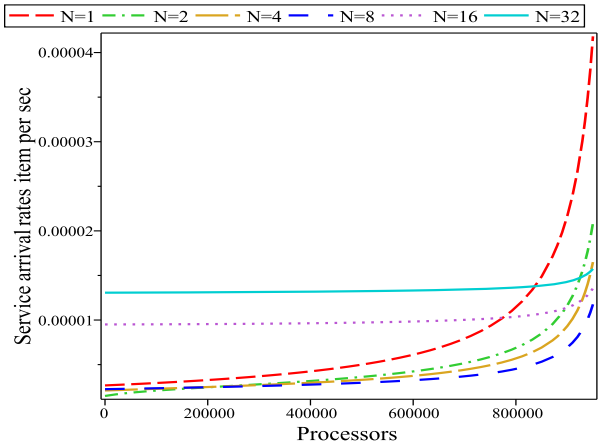
6.4 Summary

This chapter presented the results of proposed approaches in collaboration of reconfigurable processors in grid computing and Bloom filters in network processing applications.

We simulated the collaboration of four processing elements with different configurations on grid computing. Subsequently, we explore the mapping and simulate several compute-intensive multimedia kernels such as the 2D DWT and co-occurrence matrix on the proposed architecture. The experimental results show that the CRGC approach improves performance of up to $7.2x$ and $2.5x$ compared to a GPP and the collaboration of GPPs, respectively.



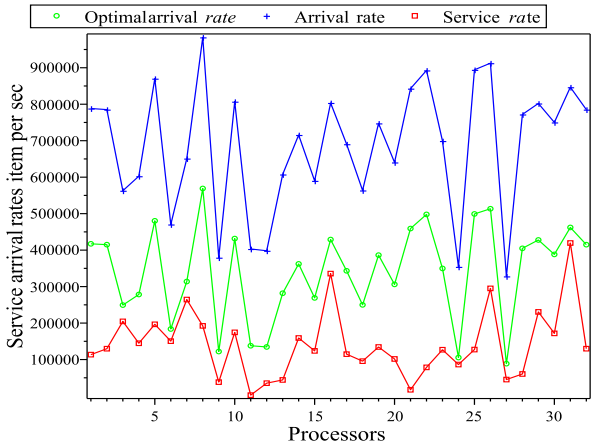
(a)



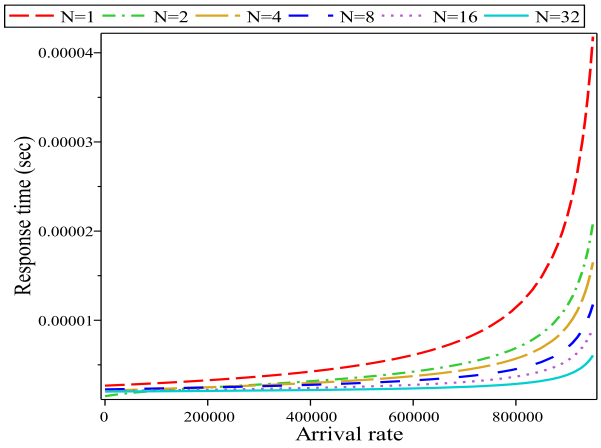
(b)

Figure 6.13: (a) Arrival/service rates and optimal arrival rate curves for different NPs. The blue and red areas show underload and overload areas. (b) Grid-oriented NP-based architectures model response time without optimal arrival rate allocation (N shows the number of NPs).

In the Bloom filters, first approach introduced a multi-level memory hierarchy and a special hardware cache architecture for counting Bloom filters that can be utilized by network processors and packet processing applications such as packet classification and distributed web caching systems. The results of mathematical analysis and implementation of CCBF for packet classification show that the proposed cache architecture decreases the number of memory



(a)



(b)

Figure 6.14: (a) Arrival/service rate and optimal arrival rates curves with optimal arrival rate allocation for different NPs. (b) Grid-oriented NP-based architectures model response time with optimal arrival rate allocation (N shows the number of NPs).

accesses when compared to a standard Bloom filter. Based on the mathematical simulation of CCBF architecture the number of accesses is decreased by at least 53%. The implementation results of software packet classifier are at most 7.8% (3.5% in average) less than correspondent simulation results. This difference is due to some parameters in the packet classification application such as: number of tuples, distribution of rules inside the tuples, and utilized hashing functions. In the second approach, we introduced a new technique to optimize memory utilization for standard Bloom filters that we call Bloom filter with an additional hashing function (BFAH). Our results show that our approach is able to reduce the average bucket size, maximum search length and number of collisions when compared to a standard Bloom filter. In the third approach, we presented a k-stage pipelined Bloom filter, the power consumption analysis and utilized a software packet classifier to customize the k-stage pipelined Bloom filter architecture in packet classification. The results of the software packet classifier with real packet traces show that more than 75% of mismatched packets can be detected by the first three stages of the pipelined Bloom filter architecture (the remaining 25% comprises 17% matched and 8% mismatched packets). Therefore, a 4-stage pipelined Bloom filter architecture with one hashing function in the first three stages and $k - 3$ parallel hashing functions in the last stage is more appropriate for power consumption optimization in packet classification.

Chapter 7

Overall Conclusions

This dissertation introduced several techniques to achieve the high-performance processing in networked and grid environments. Subsequently, the collaboration of reconfigurable processors in grid environment was presented and suitable applications such as multimedia kernels were simulated in this environment. Finally, three approaches were presented to achieve high-performance network processing using Bloom filters. The first and second techniques presented a cache architecture for counting Bloom filter and a memory optimization approach for Bloom filter using an additional hashing function (BFAH). The last one proposed a power efficient pipelined Bloom filter. Consequently, the related results was presented.

In this chapter, we present concluding remarks, major achievements and possible future directions of this research. This chapter is organized as follows. In Section 7.1, a summary of the main conclusions of this dissertation is presented. In Section 7.2, the problem statements addressed in this dissertation is revisited. In Section 7.3, the major contributions described in the dissertation are listed and finally, in Section 7.4, several possible future research directions are presented.

7.1 Summary

In Chapter 1, the importance of high-performance processing, its challenges, problem statement and solutions to achieve high-performance processing systems are presented. A motivation on the need for high-performance processing and the requirements of some high-performance processing applications was described. We presented that the modeling and design of high-performance processing systems is an ongoing research field. High-performance processing comes from parallelism. High-performance processing is possible to exploit parallelism in the applications that benefit from large scale distributed network. The parallelism available in applications in grid computing is a very attractive solution to execute either different parts of an application or several applications. Two main parts of each high-performance processing system are computing and communication resources. Furthermore, accelerating the network processing tasks for the existing computing resource in grid computing makes it more beneficial where Bloom filters are utilized in the network processing applications. Therefore, we proposed two research problems to investigate:

- How can reconfigurable computing be utilized to improve flexibility and performance in grids?
- How can Bloom filters be employed to speed up network processing in grid nodes?

In Chapter 2, we presented a general background of high-performance processing systems in networked and grid environments. Firstly, we presented the concept of grid computing, reconfigurable architectures and their capabilities. The most important capabilities of grid computing are parallel CPU capacity, exploiting under-utilized resources, and reliability while reconfigurable computing provides both performance and flexility. Therefore, the utilization of reconfigurable processors as a resource in grid computing is more beneficial to execute the computationally intensive tasks. Secondly, we discussed the high-performance network processing as an integral part of each grid system. Hence, we introduced the concept of network processors (NPs) as a main part of most network equipments e.g., routers, switches, and firewalls. A network processor (NP) is an application-specific instruction processor (ASIP) for the networking application domain with architectural features and/or special circuitry for packet processing at wire speed. Typical functions performed by network processors are the following: lookup and pattern matching, forwarding, access control and queue management, traffic shaping and control, and data

manipulation. Subsequently, we presented an overview of packet classification and Bloom filters. Bloom filter is a memory efficient and probabilistic data structure to represent a set in order to support membership checking that are utilized by many network processing applications. Additionally, we presented queuing network theory and Jackson modeling to evaluate the performance of network processing systems.

In Chapter 3, we proposed Collaboration of Reconfigurable processors in Grid Computing (CRGC) that provides both flexibility and performance to process the computationally intensive tasks. This is because, reconfigurable computing provides much more flexibility than ASIC and much more performance than GPPs. Grid computing increases the performance of computationally intensive applications by exploiting the parallelism in certain application. Each part of an application can be executed on a processing element or grid node of a large grid network.

We presented our idea to build a network of heterogeneous processing elements that are able to collaboratively work on any task that is inserted into the network on any processing element. We introduced our idea using neighborhood concept as a main part of collaboration policy that utilizes a set of primitives in a network of processing elements. These primitives implement different collaboration methods. We also analyzed a lower and upper bounds of performance for this approach. Subsequently, we explored the mapping of several computational intensive multimedia kernels such as the 2D Discrete Wavelet Transform (DWT) and the co-occurrence matrix on the proposed approach. To investigate the idea, we extend a version of grid simulator (GridSim v4) that we called the Collaborative Reconfigurable Grid Simulator (CRGridSim) to support reconfigurable processor modeling and neighborhood concept on grid environment. These multimedia kernels are packed up as a set of gridlets and these gridlets are simulated in simulation environment using CRGridSim software simulation tool.

In Chapter 4, we introduced several approaches to optimize memory and power in Bloom filters. The first approach introduced a multi-level memory hierarchy and a special hardware cache architecture for counting Bloom filters that can be utilized by network processors and packet processing applications such as packet classification and distributed web caching systems. Based on the counting Bloom filter analysis, we proposed two multi-level cache architectures (an $l\text{-}level$ and a 3-$level$ one) and subsequently presented the performance analysis. The performance metric is the number of accesses to different cache levels of the CCBF compared to the memory accesses when using the standard

Bloom filter. In the 3-level cache, we further determined the size of cache levels for optimal false positive probabilities using mathematical simulation and a software implementation. To test the CCBF concept, we implemented a software packet classifier utilizing a 3-level CCBF employing tuple spaces that are traditionally utilized in hashing systems.

The second approach introduced a new technique to optimize memory utilization for standard Bloom filters that we called Bloom filter with an additional hashing function (BFAH). The standard Bloom filter stores items from a set k times at locations pointed to by addresses that are the result of k hashing functions. The purpose of the additional hashing function is to select only one out of the k generated addresses. Consequently, it is no longer needed to store the $k-1$ redundant copies. In addition, we analyze several performance metrics (the average bucket size, maximum search length and number of collisions) for the proposed approach and compared to the standard and pruned counting Bloom filters. We implemented our approach in a software packet classifier based on tuple space search with the $H3$ class of universal hashing functions.

In the third approach, we introduced a k-stage pipelined Bloom filter architecture and the analysis of power consumption. Subsequently, we utilized a software packet classifier to customize the k-stage pipelined Bloom filter architecture in packet classification and analyzed the average number of '0's in the bit-array. Finally, we determined the packet mismatch rate for the different packet traces. Our observation of the software packet classifier for real packet traces showed that the first three stages of the pipelined Bloom filter detect most of the mismatched packets. Therefore, a 4-stage pipelined Bloom filter is sufficient to classify packets. The 4-stage pipelined Bloom filter is more appropriate than standard Bloom filter when the power consumption is critical.

In Chapter 5, we proposed a solution to optimize the arrival rate allocation between network processing elements to minimize their total response time. The solution utilized queuing network models and an optimal capacity allocation concept. Subsequently, we derived a formula to optimally allocate the arrival rate between network processors (NPs). Using this formula, the optimal arrival rate for different NPs can be evaluated to optimize response times. Furthermore, the solution was applied to a grid-oriented network processor model.

In Chapter 6, we presented the results of our proposed approaches in collaboration of reconfigurable processors in grid computing and Bloom filters in network processing. In collaboration of reconfigurable processors in grid computing, several compute-intensive multimedia kernels are studied and mapped using collaboration of four processing elements in grid computing based on neighbor-

hood policy and realistic assumptions. The experimental results show that the CRGC approach improves performance of up to 7.2x and 2.5x compared to a GPP and the collaboration of GPPs respectively.

The results of our mathematical analysis and implementation of CCBF for packet classification showed that the proposed cache architecture decreases the number of memory accesses when compared to a standard Bloom filter. Based on the mathematical simulation of CCBF architecture the number of accesses is decreased by at least 53%. The implementation results of software packet classifier are at most 7.8% (3.5% in average) less than correspondent simulation results. This difference is due to some parameters in the packet classification application such as: number of tuples, distribution of rules inside the tuples, and utilized hashing functions. We concluded from the results that incorporating a multi-level cache memory to Bloom filters will improve the performance of Bloom filter in comparison to a standard Bloom filter. Based on the simulation results of CCBF architecture the number of accesses is decreased at least by 53%. The implementation results are at most 7.8% less than corresponding simulation results.

Our results in BFAH show that our approach is able to reduce the average bucket size, maximum search length and number of collisions when compared to a standard Bloom filter.

We presented a k-stage pipelined Bloom filter, the power consumption analysis and a software packet classifier to customize the k-stage pipelined Bloom filter architecture in packet classification. The results of the software packet classifier with real packet traces showed that more than 75% of mismatched packets can be detected by the first three stages of the pipelined Bloom filter architecture (the remaining 25% comprises 17% matched and 8% mismatched packets). Therefore, a 4-stage pipelined Bloom filter architecture with one hashing function in the first three stages and $k - 3$ parallel hashing functions in the last stage is more appropriate for power consumption optimization in packet classification.

7.2 Problem Statements Revisited

The answers to the research questions presented in Section 1.2, can therefore be summarized as follows:

- How to achieve high-performance and flexible processing in networked and grid environments? High-performance computing comes from paral-

lelism in application kernels. Grid computing provides parallelism and reconfigurable processors offer both parallelism and flexibility. Therefore, we investigated to utilize both the flexibility and performance of reconfigurable processor in grid computing. The achievement of high-performance is possible using the collaboration of reconfigurable processors in grid environment. The computing capabilities of resources in such grid network can be enhanced by the collaboration of reconfigurable processors. Another main resource in grid is communication resource that is needed to process the packet in the grid. The main part of communication resources are network equipments, e.g., routers, switches, ad firewalls. These communication resources are comprised of a set of network elements called network processors (NPs). Network processors combine the flexibility of general-purpose processors with the high-performance of application-specific integrated circuits for the network processing tasks. The design of high-performance NPs improves the performance of processing element in grid.

- Can Bloom filters be employed to speed up network processing in grid nodes? Yes, the memory bottleneck due to the gap between memory and processor in the NPs that utilized in network equipments can be overcome. This is because of the proposed approaches in this dissertation decreases memory redundancies and increases the performance of memory. Therefore, a Bloom filter as an efficient data structure can be embedded in the memory architecture design in network processors and different network processing applications. In other words, the Bloom filter accelerates packet processing as an integral part of each grid node. In this way, a computing grid node can be utilized as a effective networking infrastructure provides high-performance both in terms of computing and communication. The Bloom filter accelerate the processing of received packets from grid node and provide a much more efficient memory architecture utilized in the network processors.

- How can reconfigurable computing be utilized to improve flexibility and performance in grid networks? Reconfigurable processors provide the flexibility and performance for the applications that are amenable to acceleration with reconfigurable hardware; the part of the application to be accelerated must typically exhibit a high degree of intrinsic parallelism and it must be computationally intensive. The reconfigurable processors as part of grid resources can be utilized to assists a grid node to execute the inserted tasks.

7.3 Main Contributions

In this section, the main contributions of the research described in this dissertation are highlighted:

- In order to benefit advantages of both grid computing and reconfigurable processors to achieve high-performance and flexibility, the collaborative reconfigurable processors on grid computing (CRGC) was proposed. A performance model analysis for the proposed approach to determine both lower and upper bounds was investigated. Subsequently, the collaboration has been implemented using neighborhood policy. In this policy, each processing element requests assistance only from its neighbor processing elements. The neighborhood policy is an efficient way to communicate and collaborate processing elements together. The neighborhood policy can be implemented using some primitives. A primitive is defined as a processing element with related communication link and its equipments, e.g., routers and switches, to the main processing element.

- The utilization of Bloom filter in high-performance network processing was proposed. Three approaches to make the Bloom filters more efficient were presented as follows:

  - A new technique to embed a multi-level cache memory in a counting Bloom filter (CCBF) was presented. Using the counting Bloom filter property, the number of accesses and sizes of the l-level and 3-level caches in the CCBF architecture were investigated. To realize the analysis and simulation results, a software packet classifier in basic tuple space using a $H3$ class of universal hashing functions was implemented.
  - A memory optimization technique for Bloom filter using additional hashing function (BFAH) was presented. BFAH technique operates independently from the counters and bit-array in the counting and standard Bloom filters, respectively. Analysis of the BFAH in terms of different performance metrics (e.g., average bucket size, maximum search length, and the number of collisions) compared to the pruned counting and standard Bloom filters was performed. The BFAH using a hash-based software packet classifier was implemented. The presented results showed that the BFAH approach increases performance in comparison to standard Bloom filter.

  - A k-stage pipelined Bloom filter architecture to decrease power consumption was introduced. In the bit-array of a Bloom filter, bits corresponding to the index pointed to by hashing functions are checked and a "match"/"mismatch" was determined. The match/mismatch determination process can be organized in a k-stage pipelined Bloom filter architecture. The power consumption analysis of k-stage pipelined Bloom filter was presented. The results of the software packet classifier with real packet traces showed that more than 75% of mismatched packets can be detected by the first three stages of the pipelined Bloom filter architecture (the remaining 25% comprised 17% matched and 8% mismatched packets). Therefore, a 4-stage pipelined Bloom filter architecture with one hashing function in the first three stages and $k - 3$ parallel hashing functions in the last stage is more appropriate for power consumption optimization in packet classification.

- The formulation and solution of an optimal bandwidth allocation strategy using queuing network for NP-based architectures at system level was proposed. The solution allocates optimal bandwidth between network processors in a grid-oriented environment. It encompassed a new formula based on the optimal capacity allocation concept in queuing network. The presented simulation results showed that the proposed solution is able to enhance the response time in NP-based architectures when compared to a same NP-based architectures without optimal bandwidth allocation.

7.4 Future Research Directions

This section provides future research directions and improvements to the work presented in this dissertation:

- In this dissertation, collaboration of reconfigurable processors in grid environment was investigated. The presented results show that the CRGC is useful for the computationally intensive tasks with minimum communication overheads. Several directions in this area are as follows:

  - Designing and implementing the collaboration of reconfigurable processors in grid environments based on the task-driven private clusters that we call CRGC using task-driven private cluster approach. A task-driven private cluster is a dynamic collection of

processing elements on grid that is created when a task is submitted to a processing element and after the finishing of the submitted task the processing elements (resources) are released and cluster is demolished. In the task-driven private cluster, the processing element that is owner of task called clusterhead and constructs a cluster using the cluster creation rules. Each cluster is identified by: Id, Cid, Degree and a set of states. Id and state specify an individual node, Cid shows id of cluster and degree shows the number of processing elements in the cluster. The private cluster can be constructed based on limited flooding and neighborhood concepts. To create and manage the private cluster some algorithms called cluster creation, cluster reorganization and cluster splitting algorithms are defined.

When a processing element within a private cluster submits a new task, the current private cluster is split into new clusters. The splitting process can be performed in preemption or non-preemption manners. In non-preemption policy if a current processing element is executing a task from old clusterhead, the splitting operation will suspend and wait for the processing element (PE) to finish the submitted task by old clusterhead [31], [78]. In preemption policy the current task return back to old clusterhead and the current processing element declares itself as new clusterhead and constructs new private cluster. To implement the task-driven private cluster following questions should be addressed. When the size of cluster grows the processing power is increased but the stability (life cycle) of the cluster due to splitting decreases. Based on the processing power of processing elements following research questions can arise, which size for the cluster and task is optimal? How can Gridsim be utilized? If not what is the best solution? NS or OMNeT++ simulation software tools [31], [63], [92]?

- Implementation of CRGC for some computation intensive applications such as sequence alignment in Bio-computing, and number theory using real FPGAs and reconfigurable processors.
- The performed analysis to determine the performance bounds in CRGC shows that a general performance model based on different parameters such as the number of collaborators elements, the size and number of submitted subtask, the reconfiguration time, the bandwidth of the network, the propagation delay, and scheduling algorithm can be derived. This performance model is presented

in terms of some equations (see Equation (3.2)). To derive this performance model in the simple form the related equations should be optimized. The optimization process can be performed through the optimization techniques, e.g., linear programming, integer programming.

  - A CRGC consists of reconfigurable processors and general-purpose processors while in the traditional grid only general-purpose processors are utilized. The functionality of reconfigurable processors depends on the parameters such as reconfiguration time, area, and code generation techniques. These parameters play important roles in the scheduling policies. Therefore, the scheduling of the subtasks in CRGC needs new algorithms and solutions that are different from the techniques used in the grid with general-purpose processors.

- In this dissertation, several approaches to optimize memory and power for Bloom filter were presented. The research in Bloom filter open new trends in network processing applications. Another approach in Bloom filter is an implementation of a Bloom filter using SIMD architectures. In a Bloom filter, hashing functions operate in parallel manner while in a software implementation of web caching application that is utilized in Squid Linux proxy systems these hashing functions executed in a sequential manner. Therefore, a solution to accelerate the calculation of hashing keys using on SIMD is an efficient way to get more performance.

- In this dissertation, the BFAH and CCBF approaches to increase the memory performance were proposed. The combination of both approaches increases the performance of Bloom filters. In other hands, the BFAH that utilizes a cache architecture is more beneficial than either BFAH or CCBF. This means for each bloom filter both BFAH and CCBF approaches are applied. This is because of BFAH minimizes memory redundancies and the resulting decreases the size of cache levels of CCBF.

- In this dissertation, an approach for optimal bandwidth allocation for a set of network processors was proposed. A new research trend is to make a performance model for reconfigurable processors and generalize it for the collaboration of reconfigurable processors in grid and networked environments. In other words, using multi-class queuing networks, a performance model for reconfigurable processors and their specifications is proposed. This model is a mixed queuing network (close and open queuing networks) that is based on the reconfigurable processors properties. The closed queuing network models the reconfiguration phase

of reconfigurable processors and the open queuing network models the task execution on the reconfigurable processors. Subsequently, the performance model is applied for the reconfigurable processor embedded to a general-purpose processor. Finally, a network of reconfigurable processors and general-purpose processors is modeled using a multi-class queuing network.

Bibliography

[1] "Supercomputing". http://www.filepie.us/?title=Supercomputing.

[2] "White Paper Challenges in Building Network Processor Based Solutions". www.futsoft.com/pdf/ NPwp.pdf.

[3] "NASA Collaborates with Intel and SGI on Forthcoming Petaflops Supercomputers ". http://www.climatescience.gov/default.php, May 2008.

[4] "U.S. Global Change Research Program, the Essential Principles of Climate Sciences ". http://www.climatescience.gov/default.php, March 2009.

[5] I. Adan and J. Resing. "Queuing Theory". http://www.cs.duke.edu/ fishhai/misc/queue.pdf, February 2001.

[6] M. Ahmadi, S. A. Ostadzadeh, and S. Wong. "An Analysis of Rule-set Databases in Packet Classification". In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2007*, pages 24–30, November 2007.

[7] M. Ahmadi, S. A. Ostadzadeh, and S. Wong. "Rule-set Database Inspection: Towards Data Utilization in Packet Processing". In *Proceedings of International Conference on the Latest Advances in Networks (ICLAN-2008)*, pages 122–128, December 2008.

[8] M. Ahmadi, A. Shahbahrami, and S. Wong. "Collaboration of Reconfigurable Processors in Grid Computing for Multimedia Kernels". In *Proceedings of the 5th International Conference on Grid and Pervasive Computing (GPC-2010)*, 2010.

[9] M. Ahmadi and S. Wong. "Network Processors: Challenges and Trends". In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2006*, pages 223–232, November 2006.

[10] M. Ahmadi and S. Wong. "A Performance Model for Network Processor Architectures in Packet Processing Systems". In *Proceedings of the 19th International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, pages 176–181, November 2007.

[11] M. Ahmadi and S. Wong. "Hashing Functions Performance in Packet Classification". In *Proceedings of International Conference on the Latest Advances in Networks (ICLAN-2007)*, pages 127–132, December 2007.

[12] M. Ahmadi and S. Wong. "Modified Collision Packet Classification Using Counting Bloom Filter in Tuple Space". In *Proceedings of the 25th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2007)*, pages 70–76, February 2007.

[13] M. Ahmadi and S. Wong. "A Memory-optimized Bloom Filter using An Additional Hashing Function". In *Proceedings of IEEE Globecom 2008 Next Generation Networks, Protocols, and Services Symposium*, pages 2479–2483, December 2008.

[14] M. Ahmadi and S. Wong. "An Approach for Optimal Bandwidth Allocation in Packet Processing systems". In *Proceedings of 6th Annual IEEE/ACM Conference on Communication Networks and Services Research 2008 (CNSR 2008)*, pages 208–214, May 2008.

[15] M. Ahmadi and S. Wong. "On Incorporating Reconfigurable Architectures into Grid Environments Using GridSim". In *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2008*, pages 43–50, November 2008.

[16] M. Ahmadi and S. Wong. K-stage pipelined bloom filter for packet classification. In *The 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC-09)*, pages 64–70, August 2009.

[17] D. Anderson. "Open-source Software for Volunteer Computing and Grid Computing". http://boinc.berkeley.edu/.

[18] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling". *IEEE Computer*, 35(2):59–67, February 2002.

[19] F. Baboescu, S. Singh, and G. Varghese. "Packet Classification for Core Routers: Is There an Alternative to CAMs?". In *Proceedings of 22th International Conference IEEE INFOCOM*, pages 53–63, March-April 2003.

[20] F. Baboescu and G. Varghese. "Scalable Packet Classification". *IEEE/ACM Transaction on Networking*, 13(1):2–14, February 2005.

[21] R. O. Baldwin, N. J. Davis, S. F. Midkiff, and J. E. Kobza. "Queueing Network Analysis: Concepts, Terminology, and Methods". *Journal of Systems and Software*, 66(2):99–117, December 2003.

[22] G. Bell and J. Gray. "What's Next in High-performance Computing?". *Communication of the ACM*, 45(2):91–95, February 2002.

[23] D. Benitez. "Performance of Reconfigurable Architectures for Image Processing Applications". In *Journal of Systems Architecture*, volume 49, pages 193–210, November 2003.

[24] V. Berstis. "Fundamentals of Grid Computing". http://publib-b.boulder.ibm.com/Redbooks.nsf/, November 2002. IBM Redbooks paper.

[25] B. H. Bloom. "Space /Time Trade-offs in Hash Coding with Allowable Errors". *Communication of the ACM*, 13(7):422–426, July 1970.

[26] C. Bobda and R. Hartenstein. *"Introduction to Reconfigurable Computing Architectures, Algorithms, and Applications"*. Springer, 2007.

[27] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In *Proceedings of 14th Annual Allerton Conference on Communication, Control, and Computing*, pages 636–646, October 2002.

[28] R. Buya and M. M. Murshed. "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing". *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, May 2002.

[29] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In *Proceedings of the 9th annual ACM symposium on Theory of Computing*, pages 106–112. ACM Press, 1977.

[30] F. Chang, F. Wu-chang, and L. Kang. "Approximate Caches for Packet Classification". In *Proceedings of 23th IEEE International Conference INFOCOM*, pages 2196–2207, March 2004.

[31] T. C. Chiang, P. Y. Wu, and Y. M. Huang. "A Limited Flooding Scheme for Mobile ad hoc Networks". In *Proceeings IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 473–478, August 2005.

[32] P. Clowley, M. Franklin, and H. Hamidioglu. *"Network Processor Design: Issues and Practices"*. Morgan Kaufmann, 2003.

[33] E. Cody, R. Sharman, R. H. Rao, and S. Upadhyaya. "Security in Grid Computing: A Review and Synthesis". *Decision Support System*, 44(4):749–764, 2008.

[34] S. Cohen and Y. Matias. "Spectral Bloom Filters". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2003.

[35] D. Comer and L. Peterson. *"Network Systems Design Using Network Processors"*. Prentice-Hall, 2003.

[36] K. Compton and S. Hauck. "Reconfigurable Computing: A Survey of Systems and Software". *ACM Computer Survey*, 34(2):171–210, 2002.

[37] R. W. Conners and C. A. Harlow. "Theoretical Comparison of Texture Algorithms". *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 2(3):204–222, May 1980.

[38] E. P. DeBenedictis. "Reversible Logic for Supercomputing". In *Proceedings of the 2th Conference on Computing frontiers*, pages 391–402, 2005.

[39] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. "Fast Packet Classification Using Bloom Filters". Technical Report 27, Department of Computer Science and Engineering, Washington University in St. Louis, May 2006.

[40] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. "Fast Packet Classification Using Bloom Filters". In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 61–70. ACM, 2006.

[41] B. Donnet, B. Baynat, and T. Friedman. "Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives". In *Proceedings of the ACM CoNEXT 2006 conference*, pages 1–12, 2006.

[42] C. Esteve, L. F. Verdi, and M. F. Magalh. "Towards a New Generation of Information-oriented Internetworking Architectures". In *Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–6, 2008.

[43] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area (WEB) Cache Sharing Protocol". *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[44] P. Faraboschi, G. Desoli, and J. A. Fisher. "The Latest Word in Digital and Media Processing". *IEEE Signal Processing Magazine*, pages 59–85, March 1998.

[45] S. Finch. "Transitive Relations, Topologies and Partial Orders". http://algo.inria.fr/bsolve/, 2003.

[46] White Paper for Roke Manor Research. "An Introduction to Network Processors". http://www.roke.co.uk/ download/ white_papers/ network_processors_introduction.pdf.

[47] P. J. Fortier and H. E. Michel. *"Computer System Performance Evaluation and Prediction"*. Digital Press Elsevier, 1th edition, 2003.

[48] T. Guan, E. Zaluska, and D. D. Roure. "A Grid Service Infrastructure for Mobile Devices". In *Proceedings of First International Conference on Semantics, Knowledge and Grid (SKG-05)*, pages 42–46, May 2005.

[49] D. Guo, J. Wu, H. Chen, and X. Luo. "Theory and Network Applications of Dynamic Bloom Filters". In *Proceedings of 25th IEEE International Confeence on Computer Communications (INFOCOM 2006)*, pages 1–12, April 2006.

[50] P. Gupta and N. McKeown. "Packet Classification on Multiple Fields". In *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160, 1999.

[51] P. Gupta and N. McKeown. "Algorithms for Packet Classification". *IEEE Network*, 15(2):24–32, March-April 2001.

[52] R. M. Haralick, K. Shanmugam, and I. Dinstein. "Textural Features for Image Classification". *IEEE Transaction on Systems, Man, and Cybernetics*, 3(6):610–621, November 1973.

[53] P. G. Harrison and N. M. Patel. *"Performance Modelling of Communication Networks and Computer Architectures"*. Addison-Wesley Longman, 1st edition, 1992.

[54] S. Hauck and A. DeHon. *"Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation"*. Morgan Kaufmann, 2007.

[55] D. K. Iakovidis, D. E. Maroulis, and D. G. Bariamis. "FPGA Architecture for Fast Parallel Computation of Co-occurrence Matrices". *Microprocessors and Microsystems*, 31:160–165, 2007.

[56] IBM. *"Synergistic Processor Unit Instruction Set Architecture"*, January 2007. Version 1.2.

[57] I. Kaya and T. Kocak. "A Low Power Lookup Technique for Multi-Hashing Network Applications". In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pages 179–185, 2006.

[58] I. Kaya and T. Kocak. "Energy-Efficient Pipelined Bloom Filters for Network Intrusion Detection". In *IEEE International Conference on Communications (ICC06)*, pages 2382–2387, June 2006.

[59] I. Kaya and T. Kocak. "Low-power bloom filter architecture for deep packet inspection". *IEEE Communications Letters*, 10(3):210–212, March 2006.

[60] A. Kirsch and M. Mitzenmacher. "Distance-Sensitive Bloom Filters". In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX-8)*, 2006.

[61] D. Klien. "Lagrange Multipliers Without Permanent Scarring". http://www.cs.berkeley.edu/ klein/papers/lagrange-multipliers.pdf, August 2004.

[62] K. Konstantinides. "VLIW Architectures for Media Processing". *IEEE Signal Processing Magazine*, 15(2):16–19, March 1998.

[63] M. Kozlovszky, B. Mikls, and A. Vargas. "Enabling OMNeT++-based Simulations on Grid Systems". In *Proceedings of the 2th International Workshop on OMNeT++*, 2009.

[64] A. Kumar, j. Xu, J. Wang, O. Spatschek, and E. Li. "Space-code Bloom Filter for Efficient per-flow Traffic Measurement". In *Proceedings IEEE INFOCOM*, pages 1762–1773, 2004.

[65] S. Kumar and P. Crowley. "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems". In *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS05)*, pages 91–103, October 2005.

[66] V. S. Kumar, M. J. Thazhuthaveetil, and R. Govindarajan. "Offloading Bloom Filter Operations to Network Processor for Parallel Query Processing in Cluster of Workstations". In *12th International Conference on High Performance Computing*, pages 170–179, 2005.

[67] T. V. Lakshman and D. Stiliadis. "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching". In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (ACM/SIGCOM)*, pages 203–214, 1998.

[68] E. D. Lazowska, M. K. Vernon, and J. Zahojan. "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols". In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 308–315, May 1988.

[69] Minglu Li, Xian-He Sun, Qianni Deng, and Jun Ni, editors. *"Research of Online Expandability of Service Grid"*, volume 3033 of *Lecture Notes in Computer Science*. Springer, 2003.

[70] B. Liljeqvist and L. Bengtsson. "Grid Computing Distribution Using Network Processors". In *Proceedings of the IEEE Conference on Parallel and Distributed Computing Systems*, pages 13–19, 2002.

[71] W. Lin, Z. Liu, C. H. Xia, and L. Zhang. "Optimal Capacity Allocation for Web systems with End-to-end Delay Guarantees". *Journal Performance Evaluation*, 62(1):400–416, October 2005.

[72] J. Lu and J. Wang. "Analytical Performance Analysis of Network Processor-Based Application Design". In *Proceedings of International Conference on Computer Communications and Networks*, pages 78–86, October 2006.

[73] H. W. Meuer. "TOP500 Project ". http://www.top500.org/project.

[74] H. W. Meuer. "The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience ". http://www.top500.org/files/TOP500_Looking_back_HWM.pdf, January 2008.

[75] A. Miller. "A New Way to Think About Structuring Concurrent Applications". http://www.javaworld.com/javaworld/jw-02-2009/jw-02-actor-concurrency1.html, 2009.

[76] M. Mitzenmacher. "Compressed Bloom Filters". In *Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing (PODC 01)*, pages 144–150, 2001.

[77] J. P. Morrison, P. D. Healy, and P. J. O'Dowd. "Architecture and Implementation of a Distributed Reconfigurable Metacomputer". In *Proceedings 2th International Symposium on Parallel and Distributed Computing*, pages 153–158, October 2003.

[78] S. Ni, Y. Tseng, Y. Chen, and J. Sheu. "The Broadcast Storm Problem in a Mobile Ad Hoc Network". In *Proceedings of the 5th annual ACM/IEEE International Conference on Mobile computing and networking*, pages 151–162, 1999.

[79] C. D. Ott, E. Schnetter, G. Allen, E. Seidel, J. Tao, and B. Zink. "A Case Study for Petascale Applications in Astrophysics: Simulating Gamma-ray Bursts". In *Proceedings of the 15th ACM Mardi Gras Conference*, pages 1–9, January 2008.

[80] J. D. Owens, S. Rixner, U. Kapasi, P. Mattson, and B. Towles. "Media Processing Applications on the Imagine Stream Processor". In *Proceedings IEEE International Conference on Computer Design*, September 2002.

[81] A. Partow. "General Purpose Hash Function Algorithms". http://www.partow.net/ programming/ hashfunctions/index.html.

[82] P.G. Paulin, C. Pilkington, and E. Benisiudane. "StepNP: A System-Level Exploration Platform for Network Processors". *IEEE Design and Test of Computers*, 19(6):17–26, November 2002.

[83] P. K. Pollett. "Resource Allocation in General Queueing Networks with Applications to Data Networks". In *Proceedings of the 16th National Conference of the Australian Society for Operations Research*, pages 75–90, September 2001.

[84] M. Rabbani and R. Joshi. "An Overview of the JPEG2000 Still Image Compression Standard". *Signal Processing: Image Communication*, 17(1):3–48, January 2002.

[85] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. "Efficient Hardware Hashing Functions for High Performance Computers". *IEEE Transaction on Computer.*, 46(12):1378–1381, 1997.

[86] E. Safi, A. Moshovos, and A. Veneris. "L-CBF: a Low-Power, Fast Counting Bloom Filter Architecture". In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 250–255, 2006.

[87] C. H. Sauer and K. M. Chandy. *"Computer Systems Performance Modeling"*. Prentice Hall, March 1981.

[88] N. Shah. "Understading Network Processors". Master's thesis, Berkeley University, September 2001.

[89] A. Shahbahrami. *"Avoiding Conversion and Rearrangement Overhead in SIMD Architectures"*. PhD thesis, Delft University of Technology, September 2008.

[90] A. Shahbahrami, B. Juurlink, D. Borodin, and S. Vassiliadis. "Avoiding Conversion and Rearrangement Overhead in SIMD Architectures". *International Journal of Parallel Programming*, 34(3):237–260, June 2006.

[91] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. "Implementing the 2D Wavelet Transform on SIMD-Enhanced General-Purpose Processors". *IEEE Transaction on Multimedia*, 10(1):43–51, January 2008.

[92] OMNeT++ Community Site. "OMNeT++". http://www.omnetpp.org/, 2009.

[93] M. Smith and G. D. Peterson. "Parallel Application Performance on Shared High Performance Reconfigurable Computing resources". *Performance Evaluation*, 60(1-4):107–125, 2005.

[94] H. Song. "Evaluation of Packet Classification Algorithms". http:// www.arl.wustl.edu/ hs1/PClassEval.html, 2006.

[95] H. Song, J. Turner, S. Dharmapurikar, and J. Lockwood. "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing". In *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 181–192, August 2005.

[96] V. Srinivasan. *"IP Lookup and Packet Classification"*. PhD thesis, Washington University, August 1999.

[97] V. Srinivasan. "A Packet Classification and Filter Management System". In *Proceedings of the International IEEE Conference INFOCOM*, pages 1464–1473, 2001.

[98] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification Using Tuple Space Search". In *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 135–146, 1999.

[99] W. Stallings. "Queuing Analysis". ftp://shell.shore.net/members/w/s/ws/ Support/QueuingAnalysis.pdf, 2000.

[100] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *"Wavelets for Computer Graphics: Theory and Applications"*. Morgan Kaufmann, 1996.

[101] A. Sulistio, G. Poduval, R. Buyya, and C. K. Tham. "On Incorporating Differentiated Levels of Network Service into GridSim". *Future Generation Computer Systems*, 23(4):606–615, 2007.

[102] D. Suryanarayanen. "A Methodology for Study of Network Processing Architecture". Master's thesis, North Carolina State University, 2001.

[103] M. A. Tahir, A. Bouridane, F. Kurugollu, and A. Amira. "Accelerating the Computation of GLCM and Haralick Texture Features on Reconfigurable Hardware". In *Proceedings of the International Conference on Image Processing*, pages 2857–2860, 2004.

[104] T. Takami, J. Maki, J. Ooba, Y. Inadomi, H. Honda, T. Kobayashi, R. Nogita, and M. Aoyagi. "Open-architecture Implementation of Fragment Molecular Orbital Method for Peta-scale Computing". In *Proceedings of the 2th IEEE/ACM International Workshop on High-performance Computing for Nano-science and Technology (HPCNano06)*, 2007.

[105] A. S. Tanenbaum and A. S. Woodhull. *"Operating Systems: Design and Implementation (3rd Edition)"*. Prentice-Hall, 2006.

[106] D. E. Taylor. *"Models, Algorithms, and Architectures for Scalable Packet Classification"*. PhD thesis, Department of Computer Science and Engineering Washington University, August 2004.

[107] M. A. Trenas, J. Lopez, E. L. Zapata, and F. Arguello. "A Memory System Supporting the Efficient SIMD Computation of the Two Dimensional DWT". In *Proceedings IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, pages 1521–1524, May 1998.

[108] T. Tsuei and W. Yamamoto. "A Processor Queuing Simulation Model for Multiprocessor System Performance Analysis". In *Proceedings of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 58–64, 2002.

[109] C. Ulmer, C. Wood, and S. Yalamanchili. "Active SANs: Hardware Support for Integrating Computation and Communication". In *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN 2002)*, pages 48–59, February 2002.

[110] K. D. Underwood, R. R. Sass, and W. B. Ligon. "Acceleration of a 2D-FFT on an Adaptable Computing Cluster". In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FFCM-01)*, pages 180–189, 2001.

[111] F. M. Vallina, E. Oruklu, and J. Saniie. "Distributed Processing Network Architecture for Reconfigurable Computing". In *Proceedings of the International IEEE Conference Electro Information Technology*, volume 3, page 6, May 2005.

[112] S. Vassiliadis, S. Wong, and S. Cotofana. "Network Processors: Issues and Prospectives". In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2001)*, pages 1827–1834, June 2001.

[113] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. "The Molen Polymorphic Processor'. *IEEE Transaction on Computers*, pages 1363–1375, November 2004.

[114] J. Virtamo. "Queueing Course, Complete Lecture Notes". http://www.netlab.hut.fi/ opetus/s383143/ kalvot/english.shtml, 2005.

[115] S. Wong and M. Ahmadi. “Reconfigurable Architectures in Collaborative Grid Computing: An Approach”. In *Proceedings 2th International Conference on Networks for Grid Applications (GridNets 2008)*, October 2008.

[116] C. Wu and X. Yang. “An Integrated Architecture for QoS-enable Router and Grid-Oriented Supercomputer”. In *Proceedings of 3th International Conference in Networking and Mobile Computing (ICCNMC 2005)*, pages 1218–1226, August 2005.

[117] H. Yu and R. N. Mahapatra. “A Memory-Efficient Hashing by Multi-Predicate Bloom Filters for Packet Classification”. In *INFOCOM*, pages 1795–1803, April 2008.

List of Publications

Conference Proceedings (International)

1. M. Ahmadi, and F. Nadeem, and S. Wong, "Towards the Performance Analysis of Reconfigurable Hardwares in Grid Networks", in *Proceedings of the 23th Canadian Conference on Electrical and Computer Engineering (CCECE 2010)* (Calgary, Canada), May 2010.

2. M. Ahmadi and A. Shahbahrami and S. Wong, "Collaboration of Reconfigurable Processors in Grid Computing for Multimedia Kernels", in *Proceedings of the 5th International Conference on Grid and Pervasive Computing (GPC 2010)*, (Hualien, Taiwan), May 2010.

3. M. Ahmadi, and S. Wong, "K-Stage Pipelined Bloom Filter for Packet Classification", in *Proceedings of the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 09)*, (Vancouver, Canada), August 2009.

4. A. Shahbahrami and M. Ahmadi and S. Wong and K.L.M. Bertels, "A New Approach to Implement Discrete Wavelet Transform using Collaboration of Reconfigurable Elements", in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig 09)*, (Cancun, Mexico), December 2009.

5. M. Ahmadi, and S. Wong, "A Memory-optimized Bloom Filter using An Additional Hashing Function", in *IEEE Globecom 2008 Next Generation Networks, Protocols, and Services Symposium*, (New Orleans, USA), December 2008.

6. M. Ahmadi, and S. Wong, "An Approach for Optimal Bandwidth Allocation in Packet Processing systems", in *Proceedings of the 6'th Annual IEEE/ACM Conference on Communication Networks and Services Research 2008 (CNSR 2008)*, (Halifax, Canada), May 2008.

7. S. Wong, and M. Ahmadi, "Reconfigurable Architectures in Collaborative Grid Computing: An Approach", in *the 2th International Conference on Networks for Grid Applications (GridNets 2008)*, (Beijing, China), October 2008.

8. M. Ahmadi, A. Ostadzadeh, and S. Wong, "Rule-set Database Inspection: Towards Data Utilization in Packet Processing", in *Proceedings of International Conference on the Latest Advances in Networks (ICLAN 2009)*, (Toulouse, France), December 2008.

9. M. Ahmadi, and S. Wong, "A Cache Architecture for Counting Bloom Filters", in *Proceedings of 15th IEEE International Conference on Networks (ICON2007)*, (Adelaide, Australia), November 2007.

10. M. Ahmadi, and S. Wong, "Modified Collision Packet Classification Using Counting Bloom Filter In Tuple Space", in *Proceedings of the 25th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2007)*, (Innsbruck, Austria), February 2007.

11. M. Ahmadi, and S. Wong, "Hashing Functions Performance in Packet Classification", in *Proceedings of International Conference on the Latest Advances in Networks (ICLAN 2007)*, (Paris, France), December 2007.

12. M. Ahmadi, and S. Wong, , "A Performance Model for Network Processor Architectures in Packet Processing Systems", in *Proceedings of the 19th International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, (Cambridge, Massachusetts, USA), November 2007.

Conference Proceedings (Local)

1. M. Ahmadi, and S. Wong, "On Incorporating Reconfigurable Architectures into Grid Environments Using GridSim", in *Proceedings of the 19th Annual Workshop on Circuits, Systems, and Signal Processing (PRORISC 2008)*, (The Netherlands), November 2008.

2. M. Ahmadi, A. Ostadzadeh, and S. Wong, "An Analysis of Rule-set Databases in Packet Classification", in *Proceedings of the 18th Annual Workshop on Circuits, Systems, and Signal Processing (PRORISC 2007)*, (The Netherlands), November 2007.

3. M. Ahmadi, and S. Wong, "Network Processors: Challenges and Trends", in *Proceedings of the 17th Annual Workshop on Circuits, Systems, and Signal Processing (PRORISC 2006)*, (The Netherlands), November 2006.

4. M. M. Homayounpour, and M. Ahmadi, "Automatic Transcription and Time Alignment of Persian Speech Database Using HMM", in *Proceedings of the 4th International Conference Computer Society of Iran* , (Tehran, Iran), 1998.

Samenvatting

In deze dissertatie presenteren we verscheidene technieken om hoge prestaties te behalen in netwerk en grid-omgevingen. Veel applicaties hebben een verwerkingssysteem met hoge prestaties nodig om efficient te worden uitgevoerd. Hoge prestaties stoelen doorgaans op parallelisme. De parallele aard van grid-computing is een erg aantrekkelijke oplossing om het genoemde parallelisme te gebruiken door ofwel verschillende delen van een applicatie, ofwel verschillende applicaties parallel uit te voeren. In een grid-systeem zijn computing en communicatie de meest belangrijke resources. De computing resources zijn de processoren op de knooppunten van het grid. Communicatie resources binnen een grid zijn belangrijk om taken en hun benodigde data naar de knooppunten van het grid te distribueren.

Wij stellen een innovatief platform met hoge prestaties voor om herconfigureerbare processoren te gebruiken in grid omgevingen. Daarenboven concentreren we ons op de communicatie infrastructuur en netwerkverwerkingsplatformen (verwerking benodigd voor packets) om deze te gebruiken in grid omgevingen. We presenteren de samenwerking van herconfigureerbare processoren in een grid omgeving en simuleren verscheidene rekenintensieve multimedia kernels. Vervolgens introduceren we drie manieren om netwerkverwerkingstaken te versnellen gebruikmakend van Bloom-filters in netwerk- en grid-omgevingen. De eerste twee technieken presenteren een cache-architectuur voor een Counting Bloom-filter (CCBF) en een geheugenoptimalisatie voor Bloom-filters door een extra hash-functie te gebruiken (BFAH). De derde techniek die we voorstellen is een vermogensefficiente gepipelinede Bloom-filter.

We presenteren de resultaten van onze voorgestelde technieken in samenwerking met herconfigureerbare processoren in grid-computing (CRGC) en Bloom filters in netwerkverwerkingsapplicaties, bijv. packet classificatie. De resultaten laten zien dat de CRGC aanpak de prestaties verbeteren tot 7.2x en 2.5x in verhouding tot resp. een GPP en een set van samenwerkende GPPs. De resultaten van de CCBF en BFAH voor packet classificatie tonen dat de voorgestelde technieken

het aantal geheugentoegangen verminderd in vergelijking met een standaard Bloom-filter.

Curriculum Vitae

Mahmood Ahmadi was born in Koohdasht, Lorestan, Iran on the 23^{th} of September 1973. In 1992, he graduated from Shahid Rajaei high school and at the same year he accepted as Bachelor student in Hardware and Computer Engineering in Isfehan University. He received his MSc in Computer Architectures and Engineering at the Amirkabir University of Technology (Polytechnic Tehran) in June 1997. His MSc thesis, supervised by Dr. M.M. Homayoupour, is entitled: "Automatic Alignment and Transcription of Speech Databases using Hidden Markov Models with Gaussian Mixtures". In 1998, he was offered the permanent position as lecturer in Computer Engineering Department, Razi University, Kermanshah, Iran. He worked there from 1998 till 2005.

In October of 2005, he joined the Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), Delft University of Technology, as full time PhD student under the supervision of Prof. Dr. S. Vassiliadis and Dr. Ir. J.S.S.M. Wong. His research interests include: computer architecture and engineering, Bloom filters, network processing, packet classification, high-performance processing, performance modeling, queuing theory, reconfigurable computing, and speech signal processing. He is member of IEEE, ACM and HiPEAC.