M. Faisal Nadeem

Evaluation Framework for Task Scheduling Algorithms in Distributed Reconfigurable Systems

Evaluation Framework for Task Scheduling Algorithms in Distributed Reconfigurable Systems

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 27 augustus 2013 om 10:00 uur

door

Muhammad Faisal NADEEM

Master of Science in Information Technology
Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad

geboren te Kohat, Pakistan

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. K.L.M. Bertels

Copromotor:
Dr. J.S.S.M. Wong

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr. K.L.M Bertels	Technische Universiteit Delft, promotor
Dr.ir. J.S.S.M. Wong	Technische Universiteit Delft, copromotor
Prof.dr. L. Carro	Universidade Federal do Rio Grande do Sul, Brazil
Prof.Dr.-Ing. M. Hübner	Ruhr-Universität Bochum, Duitsland
Prof.dr. J. Broeckhove	Antwerpen University, België
Prof.dr.ir. D.H.J. Epema	Technische Universiteit Eindhoven en Technische Universiteit Delft
Prof.dr.ir. E. Charbon	Technische Universiteit Delft
Prof.dr.ir. G.J.T. Leus	Technische Universiteit Delft, reservelid

This thesis has been completed in partial fulfillment of the requirements of the Delft University of Technology (Delft, The Netherlands) for the award of the Ph.D. degree. The research described in this thesis was supported in parts by: (1) CE Lab. Delft University of Technology, (2) HEC Pakistan.
Published and distributed by: M. Faisal Nadeem, Email: faisal.jaan@gmail.com

ISBN: 978-94-6186-192-4

Keywords: Simulation Framework, High-Performance Distributed Computing, Reconfigurable Computing, Task Scheduling Algorithms, Resource Management.

Printed in The Netherlands

Dedicated to

the enormity of our Universe that makes me feel so humble

&

the human curiosity that is boundless beyond all universes.

Evaluation Framework for Task Scheduling Algorithms in Distributed Reconfigurable Systems

M. Faisal Nadeem

Abstract

Recent progress in processing speeds, network bandwidths, and middleware technologies have contributed towards novel computing platforms, ranging from large-scale computing clusters to globally distributed systems. Consequently, most current computing systems possess different types of heterogeneous processing resources. Entering into the peta-scale computing era and beyond, reconfigurable processing elements such as Field Programmable Gate Arrays (FPGAs), as well as forthcoming integrated hybrid computing cores, will play a leading role in the design of future distributed systems. Therefore, it is essential to develop simulation tools to measure the performance of reconfigurable processors in the current and future distributed systems.

In this dissertation, we propose the design of a simulation framework to investigate the performance of reconfigurable processors in distributed systems. The framework incorporates the partial reconfigurable functionality of the reconfigurable nodes. Depending on the available reconfigurable area, each node is able to execute more than one task simultaneously. As part of implementation of the framework, we describe a simple mechanism for the resource information maintenance. We propose the design of data structures, which are essential parts of a **R**esource **I**nformation **S**ystem (RIS). A detailed example is provided to discuss the basic functionality of these data structures, which maintain the information regarding the reconfigurable nodes, such as their updated statuses, their available areas, and the current tasks etc.

Furthermore as a case study, we present a variety of scheduling strategies implemented to distribute tasks among reconfigurable processing nodes, utilizing the option of partial and full reconfigurability of the nodes. We propose a generic scheduling algorithm which is capable of assigning tasks to these two variants of the nodes. Using a given set of simulation parameters under the same simulation conditions, we performed various experiments. Based

on the results, it is proved that the nodes with partial reconfigurable options provide a less average waiting time per task and total task completion time. In addition, the results suggest that the average wasted area per task is less as compared to the full configuration, verifying the functionality of the simulation framework.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms and Symbols

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
GPP	General-Purpose Processor
HPC	High Performance Computing
SETI	Search for Extraterrestrial Intelligence
RPE	Reconfigurable Processing Element
CCU	Custom Computing Unit
HDL	Hardware Description Language
GPU	Graphics Processing Unit
RIS	Resource Information Service
DRMC	Distributed Reconfigurable Metacomputer
RCC	Reconfigurable Computing Cluster
RM	Replica Manager
MIPS	Million Instructions per Second
VO	Virtual Organization
PE	Processing Element
UML	Unified Modeling Language
XML	Extensible Markup Language
GPU	Graphical User Interface
LUT	Look-Up Table
SIMD	Single Instruction, Multiple Data
DS	Distributed System
DReAMSim	Dynamic Reconfigurable Autonomous Many-task Simulator
VLIW	Very Long Instruction Word
VHDL	Very High Scale Integrated Circuits Hardware Description Language

Acknowledgments

Dr. Seuss once said, "*don't cry because it's over, smile because it happened.*" Another chapter of life is over; and I choose to smile rather than cry, although with a heavy heart. No words can encompass the thankfulness that I owe to the Almighty, as written in His holy book: "*then which of the favors of your Lord will ye deny?*". I stand here smiling, only due to the efforts, prayers, and support of numerous people. I would like to thank each and everyone of them. Here goes.

Dear Stephan! I am deeply indebted to all your invaluable contributions towards the completion of my PhD studies. You epitomize the famous phrase, "*good shepherd guides gently*". You have always been readily available through all the thick and thin, during my research work and otherwise. I owe a plenty of thanks to you. I am also grateful to my promoter, Prof. K.L.M. Bertels, who has been very kind in supporting me. Thank you, Koen! I must also thank Lidwina, Eef, Erik, and Bert for their technical facilitation. My gratitude also goes to all faculty members at CE labs, especially the late Stamatis Vassiliadis whom I never met, but his inspiration is everlasting for our group. I would like to thank my PhD committee members, who have spent their precious time in evaluating my research work. I am grateful to HEC, NUFFIC, and my office at Islamabad for their funding and support.

My special gratitude goes to my dear friends and co-researchers Arash Ostadzadeh, Mahmood Ahmadi, Imran Ashraf, Fakhar Anjam, and Nadeem *bhai* for their worthwhile collaboration in my research. Dear Mahmood! I will never forget your affection and guidance like an elder brother and great friend.

I couldn't be more fortunate to have the "*Quorum*" as my office-mates; Roel, Kamana, Arash, and Mahyar. Dear Roel and Kamana! your candid talks were great. I must acknowledge that I learned a lot from you about life, faith, cultures, and the critical thought process. Dear Kamana! thanks for the "*Nepalese*" you taught me. I am truly blessed. May the spirit and warmth of our friendship continue forever.

Then, I must reveal that I had an "*office away from office*": Seyab, Motta, Mafalda '*Bibi Ji*', Innocent, Zaidi, and Kazim. You guys are wonderful people. Motta! thanks for all those frank conversations, my Dutch translations, and my client service calls to Dutch companies over the years. However, you still owe us a Moroccan dinner. God bless you. My special thanks go to my CE colleagues, '*king*' Marius, Mihai, Saleh, Said, George, Chunyang, Jae, Yao,

Roel Seedorf, Hamid, and Cuong for organizing nice social events, especially CE soccer matches. I owe distinguished acknowledgments to my Iranian friends; Behnaz, Mojtaba, Roya, Ashkan, Ghazaleh, and Mahroo. Dear Arash and Mahyar! I am half Iranian now; only due to you guys. Your culture, food, music, language, arts, and people are very close to my heart.

"*Good friends, good books, and a sleepy conscience: this is the ideal life*". I'm exalted to have good friends; plenty of them. First of all, the "*Babu Ji's*": Seyab, Fakhar, and Imran. Thanks for all those precious moments; dinners, tours, conversations, card games, shiftings, cricket, skirmishes, assistance, and much more. I also want to acknowledge my senior PhD fellows, Mehfooz, Hamayun, Laiq, Hisham, Zubair, and Tariq who always supported me like elder brothers. Then, I am indebted to the "*Popta platoon*"; Atif, Cheema, FK, Bil, Dev babu, Sandilo, Haider, Fahim, Rajab, and Sheikh. You guys are full of life and good humor. Atif! can I find another one like you? I am also thankful to numerous HEC scholars, especially Kazmi *sahib*, Ibrahim *sahib*, Umer Altaf *bhai*, Rafi *bhai*, Sharif, Mazhar *bhai*, *dr.* Atiq, Shahzad, Akram, Aleem, Sa'ad *bhai*, Haleem, Adeel, Ali Waqar, Tabish, and Aqeel, and many more. Special thanks to Umer Ijaz and Usama Malik for all the "*Delft gossips*" and PSS events. My appreciation goes to my cricket friends in Delft: Patel, Rahul, Shah Ji, Umar Naeem, Raghu, and numerous other friends that I have failed to name here.

I must also acknowledge my friend Irshad in UK, who has been my best buddy since times immemorial. Irshad! I sincerely value our friendship and frankness. Moreover, I owe a lot of commendation to all my old friends; Nowshad, Zia, Farid, Sul, Faiz, Yahya, Saif, Amin, Qaisar Khan, Wazir, Aamir Saleem, Asad Haroon, Ibrar, Adnan Safdar, Shahid Nawaz, Khalil, Ghafar Ali, Malik Akhtar, my UET Peshawar friends, and several others. I would also like to thank Mukhtar and Husnul Amin for their invaluable insights into various topics. My special gratitude goes to my teachers at school, college, UET Peshawar, and CTC at Islamabad.

Completing PhD studies is joyful for me, but one man will be more happy than anyone else in the world; my dear *abbu*. From a modest background, he has strived hard in his life to make his children successful. My parents have always sacrificed their comfort to see their children succeed. Thank you, *abbu* and *ammi*. I am grateful to my sisters Rubi and Aroosa, and my brothers Shehry and Riz, for their prayers and love. Riz! I have no words to describe my feelings about you. In simple words, I care about you more than anything. I always pray for your success and health. Thanks for all

the "*sponsored*" trips to Sweden and Italy. I would also like to give credit to Baseer uncle, Arif uncle, Shahid *bhai* and my in-laws; Haji *sahib* and his family, especially Idrees and Haroon.

Last but not the least, special thanks and admiration go to Shazia and Ahmad. Your love, care, and patience are indescribable, and beyond any words. All my prayers, wishes, and love is for you guys. Ahmad! nothing has given me more pleasure than being with you. I pray for your blissful life and successful future. Shazia! although I am the first *'victim'* of all your food experiments, they have always been wonderful and delicious. I am deeply indebted to you for all the delights and gifts you have brought in my life, especially Ahmad. God bless you both.

On my arrival in the Netherlands, I was told that—once settled—I will find it hard to re-integrate to my home country. It is so true. And, it is because of the openness of the Dutch society, particularly towards immigrants and foreigners. Dutch people are simply awesome. Always smiling and helping. The markets are diverse, copious, and abundant. The religious freedom is commendable. Netherlands characterizes a knowledge-based economy, where education is compulsory, prevalent, classless, and advanced. People are youthful, healthy, and creative; only due to their upbringing and promotive culture. Everyone rides bikes, eats cheese and potatoes, and speaks English. I will miss flowers, windmills, canals, and rainy days. In a nutshell, Netherlands will always remain as my *"home away from home"*.

I will end my acknowledgments with the following stanza from Shakespeare that summarizes life ahead:

There is a tide in the affairs of men.

Which, taken at the flood, leads on to fortune;

Omitted, all the voyage of their life

Is bound in shallows and in miseries.

On such a full sea are we now afloat,

And we must take the current when it serves,

Or lose our ventures.

M. Faisal Nadeem
Delft, The Netherlands, August 2013

1 Introduction

Recent progress in processing speeds of the computing resources, network bandwidths, and middleware technologies have contributed towards novel **H**igh **P**erformance **C**omputing (HPC) platforms, ranging from large-scale computing clusters to globally distributed systems. Consequently, most current computing systems possess different types of heterogeneous processing resources. Entering into the peta-scale computing era and beyond, **R**econfigurable **P**rocessing **E**lements (RPEs)—implemented on **F**ield **P**rogrammable **G**ate **A**rrays (FPGAs)—as well as the forthcoming integrated hybrid computing cores, will play a leading role in the design of future distributed systems. It is anticipated that the utilization of the RPEs will not only enhance the performance of these systems, but will also provide a large number of design choices in terms of programmability, flexibility, and cost-effectiveness. Therefore, it is important to develop simulation tools to comprehend these design choices, and measure the performance of reconfigurable processors in the current and future distributed systems. In this chapter, we identify what kind of research challenges are posed to develop such simulation tools. Moreover, we describe the necessary background and motivation in this regard. In Sections 1.1 and 1.2, we discuss some background on the HPC and reconfigurable computing systems. Then, we elaborate on the problem overview in Section 1.3. In the subsequent sections, we enlist the research challenges, the research methodology, and the thesis organization.

1.1 High Performance Computing (HPC) Systems

High Performance Computing (HPC) systems have been widely utilized in diverse application domains, such as climate modeling [9], bioinformatics [10], cryptography [6], drug design [11], **S**earch **F**or **E**xtraterrestrial **I**ntelligence

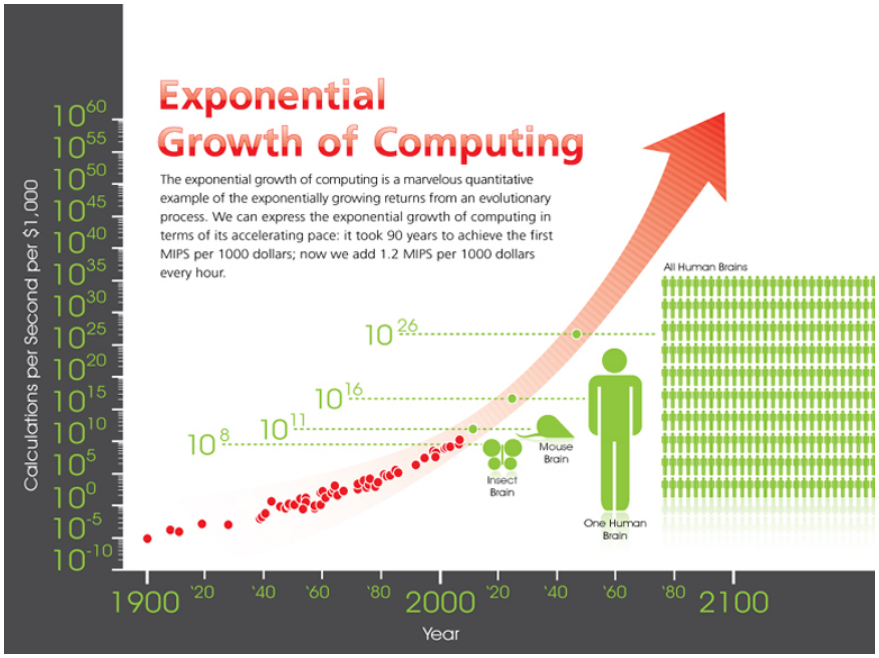


Figure 1.1: The exponential growth of computing in HPC domain. The age of petaflops (10^{15}) computing capability has approached (According to R. Kurzweil [4] in 2005).

(SETI) [12], financial modeling [13], N-body simulations [14], and many more [15]. To address the growing computational demands of a wide variety of these applications, the HPC community has provided many solutions, ranging from supercomputers, multi-FPGA platforms, customized machines and many-core architectures to general-purpose clusters, ad-hoc mobile grids, *volunteer* or *desktop* computing grids, and distributed computing grid systems [16] [17] [18] [19] [20] [21].

Figure 1.1 depicts the direction of the computing growth in the HPC domain, according to the famous book—*The singularity is near: when humans transcend biology*— by Ray Kurzweil [4]. From this figure, it can be noticed that the computing capability of the recent HPC machines is already entering into the peta-scale and beyond. It is expected that this growth will continue exponentially during the next decade.

Various solutions in terms of HPC cluster machines have been offered. Some examples of these machines are the Dell clusters, the Cascade system from Cray [22], the IBM intelligent cluster projects [23] [24], and the Solaris clus-

ter by Sun [25]. They are general-purpose in nature, and not tailored to a particular set of applications. However, they are normally proprietary, expensive, not scalable, and not customizable.

Similarly, it has been observed that the traditional supercomputing machines, such as IBM's Blue Gene—although extremely reliable and high-performance—are quite expensive, and require additional costs for their operation and maintenance [16] [26]. In contrast, the recent emergence of the *volunteer* computing systems—also known as the *desktop* grids—and the inexpensive clusters have enabled an enormous growth in the computing power to execute the above-mentioned compute-intensive applications [19]. In *volunteer* computing, unused or under-utilized processing cycles of the PC systems distributed over the Internet, are used to contribute to the computing capabilities of a grid network. These solutions are capable of processing petaflops (10^{15} floating-point operations per second) of computing power at a very low cost. The main computing element of these distributed systems are the commercially available microprocessors, such as those manufactured by Intel and AMD. These architectures might be cost-effective, flexible, and efficient, but they are **G**eneral-**P**urpose **P**rocessors (GPPs), and are originally not designed to render the required computing capacity for the growing demands of the recent HPC applications. Moreover, due to the physical limitations of the chip technology, the present-day GPP vendors are already manufacturing the multi-core processing architectures [18] [27], instead of increasing the clock speeds of a single core. With the proliferation of these multi-/many-core processors, e.g., dual-core/quad-core processors in PCs and the Cell processor in the PS3 game console, the performance of the high-performance distributed systems has incredibly increased in recent years. Subsequently, the mentioned (multi-/many-core) computing systems were further clustered in order to dedicate their processing power to a specific range of applications. However, this is different from utilizing dedicated application-specific hardware (or ASICs), which are developed for only specific applications; whereas, the processing elements in these distributed computing systems are traditionally programmable and mainly general-purpose in nature. The reason is that it is uncommon that these (clustered) computing systems are solely employed for a single application—instead, many different applications can run on these systems.

It must be evident from the discussion, that the overall performance of a distributed computing system is greatly dependent on the processing power of the employed computing elements and till now, the main processing element in these systems were (programmable) general-purpose (multi-/many-)core

processors. Nevertheless, such innovative solutions still require new programming models and design methodologies to utilize the cores in an effective manner. Therefore, it is expected that the **R**econfigurable **P**rocessing **E**lements (RPEs) as well as the forthcoming integrated hybrid computing cores, will play an important part in the design of the next-generation distributed systems. The utilization of the RPEs will not only ameliorate the performance of these systems, but will also increase the design choices in terms of programmability, power efficiency, and cost-effectiveness.

1.2 Reconfigurable Computing Systems

Advances in reconfigurable computing technology (e.g., FPGAs) over the past decade have significantly raised their interest in high-performance paradigm [28]. Venders, such as Xilinx [29] and Altera [30] offer advanced designs of FPGAs that allow the programmers to realize a particular computationally-intensive algorithm by utilizing Hardware Descriptive Languages (HDLs) like VHDL and Verilog. In this way, these devices are flexible in terms of programmability and can still offer reasonably high performance. However, until recent times, the use of FPGAs was only limited to prototyping and Custom Computing Units (CCUs), and their performance was very low, due to a limited number of programmable logic gates and inefficient CAD tools. In recent years, the advancements in the technology have enabled designers to utilize FPGAs as hardware accelerators for many applications that contain inherent parallelism. The logic gate count has significantly improved, and many advanced features, such as softcore CPUs, DSP processors, and transceivers etc. have been added. For instance, a typical FPGA of the latest Virtex-7 family from Xilinx Inc. integrates on a single chip, 1,955K logic cells, 68MB of block RAM, 3,600 DSP slices, 96 transceivers of 28 gigabits/s speed, and various other components [31].

Some of the general features of FPGAs include, programmability, power efficiency, *partial* reconfigurability, functional flexibility, extensibility (adding new functionality), (reasonably) high performance, cost effectiveness, hardware abstraction, and scalability (by adding more soft-cores). Due to these reasons, multi-FPGA systems are quite an impressive solution for high-performance and distributed computing systems [17] [32]. In many research fields of high-performance computing, FPGAs have emerged, either as custom-made signal processors, embedded soft-core processors, multipliers, prototyping designs, systolic arrays, or custom computing architec-

tures [28] [33]. With the emergence of high-level C-like programming environments, FPGAs are continuously improving [34]. However, their design complexity and utilization in a high-performance distributed system are open to innovative research. Various different approaches include efficient resource management, design-space exploration, hardware-software co-design, job scheduling algorithms, reconfigurability, and simulation and synthesis frameworks. With the prospects of including Reconfigurable Processing Elements (RPEs) in distributed systems, application task scheduling in these systems has become more pertinent and significant beyond its original scope. Similarly, new methodologies like the *partial* run-time reconfiguration for high-performance reconfigurable computing are being exploited [35]. In *partial reconfiguration,* a portion of an FPGA is dynamically modified to add new functionality, while the remaining portions continue to operate without any disruption. It allows an efficient utilization of FPGA by only configuring the required functionality at any point in time [36].

1.3 Problem Overview

These new high-performance devices are opening up novel possibilities to utilize RPEs in a distributed computing infrastructure, along with the already-existing Processing Elements (PEs), such as GPPs. This scenario can assist in a significant improvement in the overall performance of the applications. In recent years, many new computing platforms have emerged due to research and development in processing resources, network speeds, and middleware services [20] [21]. These platforms range from large-scale clusters to globally dispersed grid systems and contain diverse and heterogeneous processing resources of different number and types. These large-scale distributed computing systems are expected to possess millions of cores providing performance in peta-scale [7] [8] [37]. The growth in computing power in distributed systems has created new paradigms and opportunities for exploration of novel methods to utilize these resources in effective manners [38].

In the design of next-generation distributed systems and supercomputers, reconfigurable hardware accelerators, Graphics Processing Units (GPUs), and upcoming integrated hybrid computing cores will play a significant role [39] [40]. Moreover, GPUs, FPGAs, and multi-cores provide impressive computing alternatives, where some applications allow several orders of magnitude speedup over their General-Purpose Processor (GPP) counterpart. Therefore, future computational systems will utilize these resources to serve as their

main processing elements for appropriate applications and in some cases, as co-processors to offload certain compute-intensive parts of applications from the GPPs. Nevertheless, this scenario leads to an enormous complexity with many design and optimization alternatives for resource management, application scheduling, and run-time systems.

One important aspect of these systems is the application task scheduling. The optimal employment of resources in a distributed computing system greatly depends on how applications (and their tasks) are scheduled to be executed in the computing nodes. Various state-of-the-art scheduling algorithms for traditional grids were proposed in [41] [42] [43] [44]. Therefore, scheduling algorithms are of paramount importance for any computational systems and new ones must be developed when the characteristics of the computing nodes are changed. In their development, one must take into consideration the following aspects: static vs. dynamic computing nodes, centralized vs. de-centralized scheduling, and coarse-grained vs. fine-grained parallelism of the computing resources. The inclusion of reconfigurable hardware in the computational nodes, therefore, requires the rethinking of existing scheduling algorithms in order to take into account reconfigurable hardware characteristics, such as, area utilization, (possible) performance increase, reconfiguration time, and time to communicate configuration bitstreams, execution codes, and data.

Simulation methods have been widely used for modeling and evaluating real-world systems. The results of simulation can serve in the design of real-world systems by saving a significant amount of time and resources. For this purpose, simulation tools must be sought after to investigate the utilization of reconfigurable processors in distributed computing systems. There are numerous simulation tools for application scheduling, resource management, and resource economy in distributed systems [45] [46] [47] [48]. However, most of these tools are developed for specific purposes, and they have many limitations to allow the integration of RPEs. Therefore, it is extremely challenging to extend them for the simulation of scheduling policies to verify the performance of RPEs in distributed systems. For this reason, it is significant to design new simulation tools that can take into consideration the characteristics of RPEs.

1.4 Research Challenges

As mentioned above, the utilization of RPEs in distributed computing systems is unfolding new possibilities to assist in a significant performance improvement of the applications. The resultant growth in the computing power in distributed systems has created the opportunities for the exploration of novel methodologies. However, this scenario has also posed new research challenges and open questions. In the following, we outline the challenges that are addressed in this dissertation.

Challenge 1—Can the existing simulation tools in the High Performance Computing (HPC) and the distributed computing domains efficiently allow to integrate the novel processing elements, such as RPEs?

There are many state-of-the-art simulation tools (such as, Gridsim [45], Bricks [46], MicroGrid [47], and SimGrid [48]) for the resource management and application scheduling in such systems. The most notable among these simulators is Gridsim [45]. It is a powerful toolkit which enables users to model and simulate the characteristics of computing grid resources and allows to do experiments regarding scheduling policies, grid economy, and advance resource reservation. It provides an extensive framework, which allows modeling of grid resources consisting of GPPs. The computing capability feature of resources in Gridsim is modeled in terms of their MIPS ratings, only specific to GPPs. Each resource is modeled by specifying its number of machines and the processing elements (PEs) contained in each machine. Once defined by the user, the resources in Gridsim have fixed computing capabilities during a simulation run. Due to their fixed computing capacity, their specification can not be changed to add reconfiguration parameters which require a dynamic organization of resources during simulation. Furthermore, the fixed structure of resource management system in Gridsim is insufficient to accommodate the dynamic nature of reconfigurable processors. Hence, extending Gridsim for modeling reconfigurable processors is extremely difficult in its current implementation state.

Challenge 2—What parameters differentiate RPEs from GPPs? Are there any previous attempts to integrate RPEs governed by these parameters in the existing simulation tools?

In order to analyze the possibility of incorporating RPEs in a simulation tool, it is important to enlist the parameters that distinguish the RPEs from GPPs. We outline these parameters in Chapter 2. Previously, some attempts (e.g.,

CRGridsim [49]) were made to extend Gridsim to add RPE functionality but they only include speedup factor of a reconfigurable element over a GPP, as the primary reconfiguration parameter. Other important parameters, such as reconfigurability, area utilization, reconfiguration method (partial or full), reconfiguration delay, hardware technology, application model, and application size were ignored. These parameters play a key role in modeling the RPEs in distributed computing systems.

Challenge 3—What evaluation metrics are the most important to illustrate the performance of such tools?

Apart from measuring the performance or speedup of a certain application, what other metrics are required to test the efficacy of the simulation tools allowing to model and simulate RPEs. These metrics include the *reconfiguration delay* measurements, the *average wasted reconfigurable area* of the nodes, the *waiting time per task*, and the *simulation workload* etc. We identified and discussed these performance metrics in Chapter 5.

Challenge 4—What kind of dynamic data structures are required in order to perform the resource management?

In a dynamic environment, the behavior of a computing system changes with time. In order to model and simulate a system containing RPEs, it is important to investigate what new methodologies are required to manage the computing resources. Moreover, what type of data structures are needed to maintain the information services corresponding to these resources. In this respect, we proposed a simple **R**esource **I**nformation **S**ystem, which is described in detail in Chapter 4.

Challenge 5—What are the scheduling strategies to assign tasks to a distributed computing system containing RPEs?

Since a typical scheduling system must deal with new parameters while assigning tasks to RPEs, the scheduling strategies need to be developed accordingly. For instance, a task scheduler can send a task to a particular RPE, only if it contains sufficient reconfigurable area. For this reason, the scheduler must be aware of the area on all the RPEs to make a mapping decision. In another scenario, a scheduler can assign multiple tasks to an RPE, only if it allows *partial reconfiguration* mechanism. Consequently, new scheduling strategies are required for a distributed computing system containing RPEs.

Challenge 6—What are the possible virtualization models to add RPEs in an existing distributed computing system?

Virtualization allows multiple application tasks to utilize resources at a given

time, by adding abstraction layer between tasks and resources. Traditional distributed systems are virtualized for the GPPs only, and new models are required to virtualize RPEs in existing systems. In this scenario, it is important to propose a virtualization model for a generic computing node, that can integrate both GPPs and RPEs.

Challenge 7—What is the impact of using *partial* reconfiguration as compared to *full* reconfiguration, on the performance of scheduling strategies in a distributed system with RPEs?

What is the impact on the overall performance of the system, if the nodes allow to execute multiple tasks as compared to a single task at a given time. Subsequently, it is interesting to investigate what kind of scheduling strategies are required in both these cases.

1.5 Research Methodology

In this section, we describe the research methodology carried out to address the research challenges and open issues mentioned in the previous sections. The main goal of the dissertation is to develop a simulation framework that can assist in testing scheduling strategies for RPEs in distributed systems. The following steps outline our research methodology:

- Investigate and enlist the important parameters that differentiate the RPEs from the general-purpose processors. These parameters serve as a basis to ascertain what policies are needed to schedule application tasks to RPEs in distributed systems.
- Determine what formulation is required in order to model reconfigurable nodes, processor configuration, and application task in distributed computing systems.
- Perform a survey of several distributed computing projects, middlewares, schedulers, and simulation tools. Outline the main features of these tools to compare their pros and cons, in order to check the possibility of extending them for the integration of RPEs.
- Investigate various scheduling proposals and resource management architectures in distributed computing.
- Based on the investigation of important parameters and existing simulation tools, introduce a new simulation framework that enables

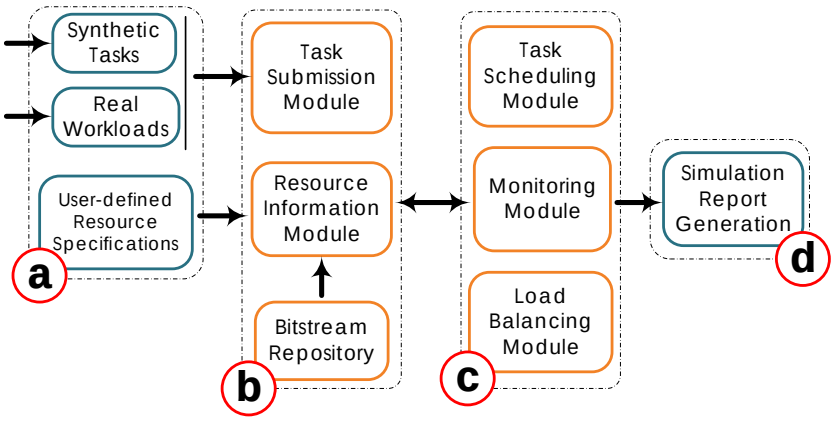


Figure 1.2: An overview of the DReAMSim simulation framework, where (a) represents the *Input subsystem*, (b) the *Resource Information subsystem*, (c) the *Core subsystem*, and (d) the *Output subsystem*.

scheduling of application tasks in distributed systems containing RPEs.

Based on above-mentioned steps, we propose a novel simulation framework that allows to model and simulate complex reconfigurable resources, processor configurations, and application tasks [50] [51]. The simulation framework is termed as **D**ynamic **Re**configurable **A**utonomous **M**any-task **Sim**ulator (DReAMSim), and it incorporates various simulation parameters required to simulate reconfigurable processors. Moreover, it implements a resource information system that helps to maintain the resources in the system. It is different from the traditional simulation tools, because the characteristics of the nodes are dissimilar to GPPs. The framework provides a basis to investigate different scheduling policies for a given set of parameters. Subsequently, it implements the partial reconfigurability feature to the computing nodes, where a node region can be reconfigured dynamically, while the other regions of the node are running tasks. The development of framework also entailed to investigate what type of scheduling policies are required to assign tasks to reconfigurable computing nodes in a distributed system. For this purpose, we introduced a generic scheduling process that takes into account the parameters regarding the reconfigurable nodes.

Figure 1.2 depicts an abstract view of the DReAMSim framework, along with its various subsystems. The framework allows a user to utilize the *user-defined resource specification* module to model and simulate reconfigurable

nodes, configurations, and application tasks. Subsequently, a typical scheduling strategy can be tested for different simulation conditions using the *task scheduling module*. The detailed design and implementation of the DReAMSim are discussed throughout this dissertation.

Finally, we propose a generic virtualization framework similar to workflow management systems that enable the match-making between application tasks and reconfigurable resources.

1.6 Dissertation Organization

This dissertation is organized in several chapters. Mainly, it describes various aspects of the design and development of a simulation framework for the utilization of RPEs in distributed systems. An outline of this dissertation is given in Figure 1.3, which depicts the relation between the research challenges, dissertation contributions, and various chapters,. In the following, we summarize the contribution of each chapter.

Chapter 2—Reconfigurable Computing in Distributed Systems

Chapter 2 describes an overview of various simulation tools in the distributed computing domain. We discuss the merits and limitations of these tools, and discuss the processing elements that can be modeled and simulated. Furthermore, we give a survey of different real computing systems and projects, that are utilizing the reconfigurable processors. They include multi-FPGA systems, grids utilizing FPGAs, and customized HPC systems containing FPGAs. The chapter also provides a brief overview of scheduling algorithms used in distributed computing domain.

Chapter 3—Virtualization of RPEs in Distributed Systems

This chapter proposes a general virtualization framework for RPEs and application tasks in distributed computing systems. First, we discuss various use-case scenarios of the utilization of RPEs in different application domains. Secondly, we propose a virtualization model for a computing node, which incorporates both General Purpose Processors (GPPs) and RPEs. Thirdly, we discuss a model for a typical application task, which needs a particular processing element for its execution. Finally, we conclude the chapter with a case study of an application from the bioinformatics domain, that requires various types of processing elements in a distributed system. The case-study provides all the possible mapping options of different tasks in the application, based on the use-case scenarios.

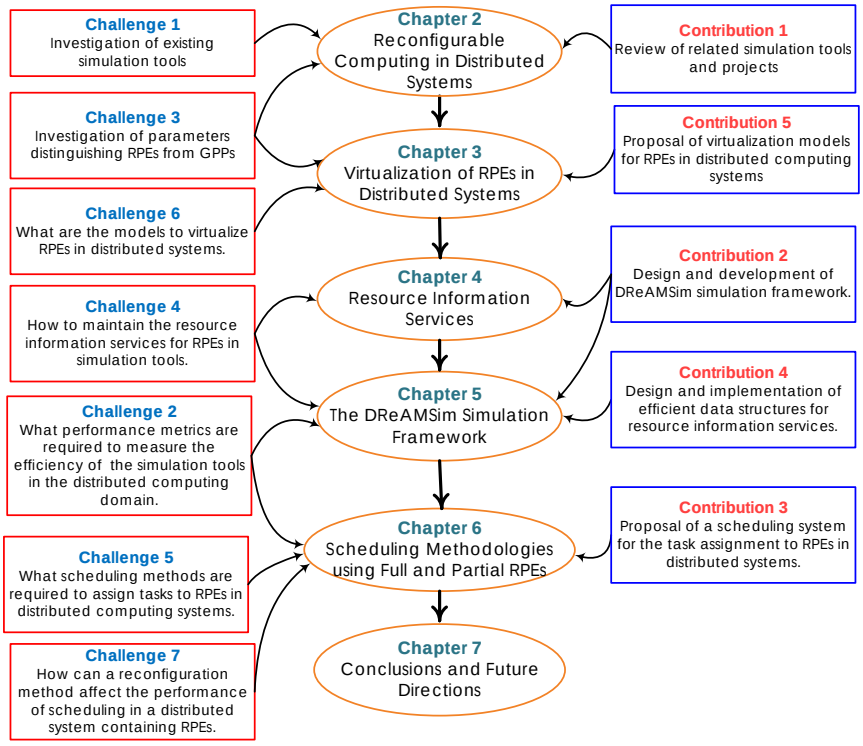


Figure 1.3: The dissertation organization, depicting the relation between various chapters, the research challenges, and the dissertation contributions.

Chapter 4—Resource Information Services

Resource management is one of the fundamental components in distributed computing systems. In this chapter, we depict several scenarios of resource management in a distributed system with RPEs. Some basic concepts are discussed in this regard, and we give a formal system model. Moreover, we describe a simple mechanism for the resource information maintenance. We propose the design of data structures, which are essential parts of a Resource Information System (RIS). These data structures maintain the information regarding the nodes, such as their updated statuses, their available areas, and the current tasks etc. The chapter also contains a detailed motivational example to elaborate the basic functionality of the proposed data structures.

Chapter 5—The DReAMSim Simulation Framework

In this chapter, we describe the details of the DReAMSim framework. First,

we describe the implementation of the RIS data structures, followed by the UML model of the framework. Then, we discuss the various performance metrics and statistics generated by the simulator. We also describe the importance of these performance metrics. We conclude the chapter by presenting a case study of a scheduling algorithm, and giving various simulation results generated by the simulator.

Chapter 6—Scheduling Methodologies using *Full* and *Partial* Reconfigurable Nodes

This chapter presents a variety of scheduling strategies implemented to distribute tasks among RPEs, utilizing the option of *partial* and *full* reconfigurability of the nodes. We propose a generic scheduling algorithm which is capable of assigning tasks to these two variants of the nodes. Using a given set of simulation parameters, and the same simulation conditions, we performed various simulation experiments. Based on the results, it is proved that the RPEs with *partial* reconfigurable options provide a *less average waiting time per task* and *total task completion time*. In addition, the results suggest that the *average wasted area per task* is less as compared to the *full* configuration, verifying the functionality of the simulation framework.

Chapter 7—Conclusions and Future Work

In this chapter, we give several conclusions drawn from the dissertation. We also enlist a set of possible extensions suggested for the simulation framework. Finally, we provide an insight into the future directions based on this thesis.

The contents of this dissertation are based on a number of peer-reviewed publications. At the end of each chapter, we enlist the relevant publication(s). Moreover, a complete list of all publications is given at the end of the dissertation.

2

Reconfigurable Computing in Distributed Systems

In this chapter, we provide a detailed background information related to this dissertation. We describe various distributed computing projects, middleware technologies, real testbeds, schedulers, and simulation tools. We discuss some systems that utilize reconfigurable processors in distributed computing paradigm. Furthermore, we give a comparison between important simulation tools, and describe why it is necessary to introduce a novel simulation framework in order to incorporate reconfigurable nodes in distributed computing systems, to evaluate new scheduling algorithms. Finally, we describe a set of simulation parameters that must be considered in order to develop such a simulation tool.

2.1 Introduction

In recent decades, high-performance distributed computing systems have emerged as powerful platforms to execute diverse and compute-intensive applications in various application domains. These systems include world-wide grid computing systems, supercomputers, clusters, multi-FPGA platforms, and *volunteer* computing systems [16] [17] [18] [19] [20] [21]. In real-world, the design and deployment of these systems can be extremely complicated. Furthermore, it is difficult and expensive to estimate their performance to execute applications and their scheduling on real systems. Apart from real testbed experiments, the research in distributed systems normally focus on analytical studies and simulation.

Analytical performance modeling—including, the time complexity of algorithms, the queuing modeling techniques, and the HW/SW partitioning

etc.—is a valuable methodology often used to determine the performance and operating measures of a real-world system [52] [53] [54]. Although, these modeling techniques can assist in discovering basic behavioral theorems, their set of assumptions are too simple to comprehend all the required parametric conditions of a real system. Moreover, in order to convince the designers, the analytical results are always validated by proper experiments under the same set of parameters and conditions. These experiments must be performed on either small customized setups or on large real systems. In case of customized setup, the experimental results are controlled and repeatable. But they cannot be representative to the real-world scenarios. Whereas, it is impractical to use real systems for validation purpose, as their access is difficult to obtain and the experiments are not repeatable and uncontrollable, due to the dynamic nature of computing resources. In both cases, it is still very hard to validate the results under all parametric conditions.

On the other hand, simulation of a real-world system can resolve many of these problems. Simulation is a widely-used methodology employed to simulate a large-scale system before its actual deployment. In this way, the designers can save a significant amount of time and resources. A carefully developed simulation can provide a detailed study of a real system, and help in analyzing its various scenarios, such as performance, behavior, and viability. In this chapter, we describe a detailed analysis of various tools developed for the simulation of distributed systems.

The remainder of the chapter is organized as follows. Section 2.2 introduces high-performance computing and describes *volunteer* computing as an example. It also discusses some real testbeds and important middleware technologies in distributed computing. In Section 2.3, we briefly discuss reconfigurable computing, and the benefits of *partial* reconfiguration. In Section 2.4, we give a detailed description of various projects, where reconfigurable processors are utilized in a distributed system. Section 2.5 presents an overview of the application scheduling in distributed systems, and briefly discuss two famous grid schedulers. In Section 2.6, we give a detailed description of different simulation tools in distributed computing paradigm. Moreover, we give a comparison between these tools. In Section 2.7, we discuss the additional simulation parameters, that are required to incorporate reconfigurable nodes in a simulation tool. We also introduce our proposed simulation framework. Finally, we give summary and conclusions in Section 2.8.

Name	Peak speed (in PetaFLOPS)	Location
Titan (Cray)	17.6	Oak Ridge National Laboratory (ORNL), USA
Sequoia (IBM)	16.3	Lawrence Livermore National Laboratory, USA
K-computer (Fujitsu)	10.5	RIKEN Computational Science Institute, Japan
BOINC projects combined	8.8	Distributed

Table 2.1: A performance comparison between BOINC projects and the world's fastest supercomputers [1] [2].

2.2 High-performance Distributed Computing

Growing demands of computational power and flexibility are resulting in systems which are more complex to integrate. It has become hard to merge many different design alternatives, large number of possible configurations, and the demand for additional functionalities in a system at the design time. Large-scale high-performance and distributed computing systems have been developed to offer reliable, pervasive, ordered, and inexpensive access to computing resources which are dispersed over the Internet [20] [21]. Many applications are utilizing these computing systems, and they include weather forecasting [9], drug designing [11], bioinformatics and system biology [10], physics [14], and financial modeling [13] etc. These computing systems comprise of the traditional supercomputers [16] [26], computational grids [20] [21], *volunteer* or *desktop* computing systems [19], multi-core processing platforms [18], multi-FPGA architectures [17], and dedicated cluster machines [23] [22].

2.2.1 *Volunteer* Computing

Volunteer or *desktop* grid computing is an enabling environment, where the Internet users—termed as the *volunteers*—dedicate the *idle* computing cycles of their desktop PCs to large-scale projects that cannot be handled by a single-bounded machine [55]. In this way, a powerful virtual computing machine can be acquired. The project was initiated at UC Berkeley, and it provides an open-source middleware infrastructure to set up the *volunteer* com-

BOINC project	Host organization	Application domain
SETI@Home	UC Berkeley, USA	Astronomy
World Community Grid	IBM, USA	Medicine
Rosetta@Home	Washington university, USA	Biology
Einstein@Home	Wisconsin university, USA	Astronomy
Climate Prediction	Oxford university, UK	Earth Sciences
MilkyWay@Home	RPI, Rensselaer, USA	Astronomy
LHC@Home	CERN, Europe	Physics
Malaria Control	Swiss Trop. Institute, Europe	Medicine
PrimeGrid	R&D Hessen, Europe	Mathematics
Proteins@Home	Ecole Poly, France	Biology
ABC@Home	Leiden university, Netherlands	Mathematics

Table 2.2: Most active current BOINC projects in terms of the number of users (http://boincstats.com [1]).

puting projects. Since the first BOINC-based scientific project commenced in 1995, there are more than 50 active projects today [1]. The most notable project is *SETI@Home*, which was launched to **S**earch for **E**xtra **T**errestrial **I**ntelligence (SETI) outside our planet [12]. As of March 2013, this project demonstrates the enormity of computing power obtained through a *volunteer* computing setup, by providing 581 TeraFLOPS. Table 2.1 presents a comparison—based on peak performance in PetaFLOPS—between the world's fastest supercomputers and BOINC-based projects [1] [2]. This comparison shows how BOINC projects are producing a computational speed, which is comparable to the fastest supercomputers in the world.

Volunteer Computing Projects

Many large-scale scientific problems can be easily broken into smaller independent tasks in a *volunteer* computing environment. These tasks are assigned to each BOINC client for processing. Table 2.2 presents a list of major *volunteer* computing projects in terms of number of active users.

SETI@Home is the largest BOINC project in terms of number of users (more than 2.5 million users [1]) and computational performance [12]. The main purpose of *SETI@Home* project is to search for the possible evidence of radio signals from the extraterrestrial intelligence. For this purpose, the project takes data from the Arecibo radio telescope, and creates Discrete Fourier

Project	Host organization	Application domain
BioGrid [57]	BioGrid Kansai, Japan	Drug discovery
EGI [58]	European Union	Earth Sciences
NorduGrid [59]	Oslo University, Norway	High Energy Physics
XSEDE [60]	University of Illinois, USA	Astronomy, Physics etc.
Virtual laboratory [11]	Melbourne University, Australia	Molecular Drug Design

Table 2.3: Important production grid projects [3].

Transform (DFT) based computational tasks. These tasks are analyzed by using a software that searches for meaningful intelligent signals. Each user installs the software on its PC, receives data from the SETI project, and processes the tasks independently.

Another project—initiated by Oxford university in the UK—utilizes computer simulations to reduce the volatility in the climate prediction models [56]. The project is known as the *Climate Prediction*, and it uses various weather model ensembles and parameters to run large number of simulations, assisted by BOINC platform. Similarly, *Protein@Home* is another *volunteer* computing project to test and evaluate new algorithms and methods concerning the prediction of protein structure from sequence [1]. Other projects, such as the PrimeGrid make use of BOINC platform to search for large numbers with certain properties, for instance, searching for prime numbers.

2.2.2 Major Production Grid Projects

Apart from the *volunteer* computing, there are many production grid projects working on several different applications. Table 2.3 enlists some important projects and their application domains. In the following, we briefly discuss two such project.

The **E**xtreme **S**cientific and **E**ngineering **D**iscovery **E**nvironment (XSEDE) [60] is a comprehensive distributed computing infrastructure for open scientific research. It enables a single virtual environment that can be utilized by scientists to aggregate their computing resources, scientific tools, and collections of data. The grid system targets applications in the various fields such as astronomy, chemistry, earthquake mitigation, geophysics, global atmospheric research, neuroscience, molecular biology, cognitive science, physics, and seismology. A researcher can get access to the grid by using command

line through web portals to design its own workflows and access the desired resources. Similarly, the **E**uropean **G**rid **I**nfrastructure (EGI) is an extensive collaboration between various resource providers, organizations, and institutes across Europe to cooperate in sharing resources, scientific results, and tools [58].

2.2.3 Important Middlewares

A *middleware* is a software infrastructure, that is responsible for providing resource coupling services in order to aggregate computing resources to set up a large-scale distributed computing system, which is often termed as a **V**irtual **O**rganization (VO). These services include information security, process management, access to resources and Quality of Services (QoS). Table 2.4 enlists core middleware efforts in the distributed computing.

Globus Toolkit

Globus is the most extensive open-source software toolkit, that is developed and maintained by the Globus Alliance [21]. It is used to enable the development of a computational grid by compounding heterogeneous and distributed resources into a single virtual organization. It provides a set of modular components that allow basic grid services like resource allocation and management, security and authentication, network management and job submission. Essentially, it is a set of libraries and programs that address common problems when constructing a distributed grid system. It is built as a layered software architecture where high level global services rely on low level local services. Globus supports wide range of applications and programming models to incorporate to meet the specific needs of a user. Globus services have well-defined interfaces in the form of **A**pplication **P**rogram **I**nterfaces (APIs), that can be utilized by the applications. These APIs include security infrastructure, information services, resource allocation, discovery and failure mechanisms, and file storage. The toolkit offers a modular "*bag of technologies*" and enables incremental development of grid-enabled tools and applications.

Legion

Legion [63]—developed at university of Virginia, USA—is an object-based middleware framework, that facilitates the seamless and transparent interac-

Project name	Host institution	Description
Globus [21]	University of Chicago, USA	The most extensive open-source software toolkit for building grids.
GridBus [61]	University of Melbourne, Australia	A toolkit mainly used for job scheduling on grid resources.
UNICORE [62]	Jülich Supercomputing Centre (JSC), Germany	Java-based environment for distributed computing.
Legion [63]	University of Virginia, USA	An object-based, metasystem software.
ARC [59]	NorduGrid, Oslo University, Norway	Advanced Resource Connector (ARC) is open-source middleware.

Table 2.4: Major middleware technologies in distributed computing.

tion of heterogeneous distributed machines to develop a metasystem, in such a way that the end users view the whole metasystem as a single virtual machine. In Legion system, hardware and software resources are represented as *objects* that can respond to the methods invoked by the other *objects* in the metasystem. An API is defined by Legion for the *object* interaction rather than a communication protocol. The *object* instances are managed by classes which contain methods. The class *objects* are provided system-level capabilities. They can create, define, manage, activate, deactivate and schedule new *object* instances. Class *objects* can also provide state information to the client *objects*. Legion enables the users to define their own classes or redefine or override certain functionalities in an already existing class. This means that users can use Legion according to their own needs. Set of methods of a certain *object* define an interface to the other objects of a Legion metasystem. The object-oriented nature of Legion makes it suitable for the design of complex distributed metacomputing networks.

UNICORE

UNiform **I**nterface to **CO**mputer **RE**sources (UNICORE) is a middleware project, financed by the German ministry of education and research [62]. In UNICORE, the user is provided with a uniform interface for the development of distributed applications that can be executed on supercomputer resources in a seamless manner. Applications are divided into multiple UNICORE jobs that can be executed on different distributed computing systems. A UNI-

CORE job is multi-part application containing the information about remote resources requirement, inter-parts dependencies and address of the destination system. It can be viewed as a recursive object that contains tasks and job groups. Job groups themselves contain other job groups and tasks. These job groups hold destination system's address to run the included tasks. UNICORE offers a ready-to-run distributed system including client and server software. It provides distributed processing and data resources in a seamless and secure way through standard web interface and Java applets.

2.3 Reconfigurable Computing

During the last decade, the proliferation of reconfigurable computing devices—most commonly used are FPGAs—has increased the prospects of their utilization in high-performance computing systems [33] [28]. Recent advancement in FPGA designs offer more flexibility and performance for various compute-intensive applications. Due to the increase in logic gates and other advanced features, the current FPGA devices are being utilized as hardware accelerators [17]. Traditionally, the FPGAs have been exploited in three different approaches.

- Connecting an FPGA board to a general-purpose desktop PC through the PCI interconnect. The host PC runs software drivers that allow the configuration of FPGA according to the application algorithm and exchange data through the PCI bus [5] [64] [65].

- Developing a custom motherboard that integrates multiple FPGAs together through a high-speed interconnect system. Then a particular algorithm is mapped on the system, to exploit the fine-grained parallelism between FPGAs [17] [32].

- Utilizing the FPGA device as a functional unit coupled to a microprocessor. In this architecture, the microprocessor operates in a sequential von Neumann style during the execution of the application, but it *offloads* some compute-intensive part(s) of the algorithms to the hardware functional unit. One example of such an architecture is the MOLEN polymorphic processor. It couples a general-purpose microprocessor to a reconfigurable hardware logic unit—serving as a **C**ustom **C**omputing **U**nit (CCU)—that is capable of accelerating compute-intensive kernels found in a certain application [66].

With the emergence of new high-level programming tools, the FPGA technology is continuously improving [34]. New research paradigms are opening up in various fields, such as design-space exploration, hardware/software co-design, and simulation and synthesis tools.

2.3.1 What are *Partial* Reconfigurable Systems?

Partial reconfiguration is the capability to reconfigure select regions of an FPGA, after its initial configuration [67]. It is possible to perform partial reconfiguration on run-time, while the device is already active, and this approach is called *dynamic* run-time partial reconfiguration. On the other hand, it can also be performed, when the device is shut down—termed as the *static* partial reconfiguration. There are several advantages of partial reconfigurable systems [68] [36]:

- Multiple applications can be executed on a single FPGA device. In case of *dynamic* partial reconfiguration, the device can continue to operate without any disruption, and there is no loss of performance.
- Due to high-speed networks and efficient interconnects, there is an emerging trend of utilizing FPGAs in distributed systems. Due to partial reconfiguration services and vender support, these devices can be controlled and modified from remotely.
- Due to partial reconfiguration technology, it is possible to execute multiple applications on a single device. Therefore, multiple users can share a single device in a virtual environment. It can reduce power consumption and costs, and increase the efficiency of the device.
- Since the approach allows to make modifications to a certain portion of the device, instead of the entire device, the reconfiguration time is significantly reduced. It is because the reconfiguration time is directly proportional to the configuration bitstream size.

In recent times, the partial run-time reconfiguration is being exploited in high-performance computing systems [69] [35]. With the above-mentioned benefits, this technology can play an important role to increase the overall performance of a computing system.

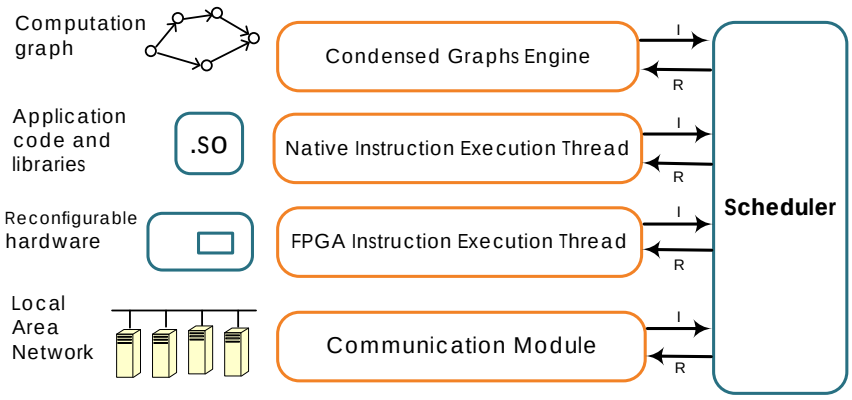


Figure 2.1: The DRMC computation process in terms of the flow of **I**nstructions (I) and **R**esults (R) [5].

2.4 Distributed Reconfigurable Projects

The promise and growth of the computing capability of reconfigurable hardware processors have opened new opportunities to utilize these resources in the large-scale distributed computing systems. Consequently, the performance of many applications can be enhanced considerably. In recent years, many distributed computing projects have already started utilizing these resources [39] [40] [38] [70]. In this section, we survey several previous works in which the reconfigurable processors are utilized in grid systems, cluster computing paradigm, and customized systems.

2.4.1 The Distributed Reconfigurable Metacomputing (DRMC)

Metacomputing is defined as, the sharing and utilization of powerful computing resources transparently available to the user via a networked environment. In [5], the architecture and implementation of a Distributed Reconfigurable Metacomputer (DRMC) are demonstrated in which, the applications are executed on a cluster that contains FPGA nodes. Figure 2.1 depicts the design of the metacomputer. The applications are modeled in the form of condensed graphs of computation. The inherent parallelism in these applications is represented in terms of graph sets that generate instructions to be executed on various nodes. The metacomputer is implemented as a peer-to-peer UNIX application composed of a daemon and a multi-threaded computation process.

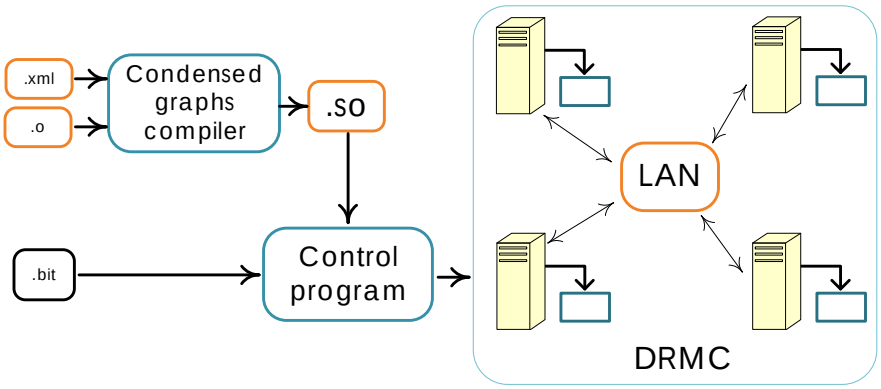


Figure 2.2: The DMRC application development process [5].

Each node runs the daemon and listens for the incoming messages from other nodes. When it receives a message, it starts the computation process by scheduling tasks (instructions) to different modules in the native/initiator node or a remote node depending upon the load conditions of different nodes. Each node publishes its load conditions to the others. If all the nodes are heavily loaded, the scheduling of a task is throttled. Configuration bitstreams are transmitted to the FPGAs communicate through PCI bus interface.

Figure 2.2 depicts the application development process. A DRMC application consists of an XML definition file which contains a set of graph definitions, a set of executable instructions and a set of user-defined types. Instruction are either object codes (.o files) or FPGA configurations (.bit files) or both. The condensed graphs compiler is an application that compiles the set of instructions needed by the application to produce a shared object (.so) file ready for dynamic linking by the metacomputation functions created automatically by the compiler required to register the graph, instruction and type definitions with the metacomputer. Any FPGA configurations required by the computation are loaded separately by the metacomputer when needed. In [71], DRMC is utilized to implement the famous cryptographic RC5 key-crack algorithm [72].

2.4.2 The Distributed Reconfigurable Computing Project

In [6], the distributed reconfigurable hardware is utilized to break the RC5 algorithm [72] by distributing generic VHDL code implementations to differ-

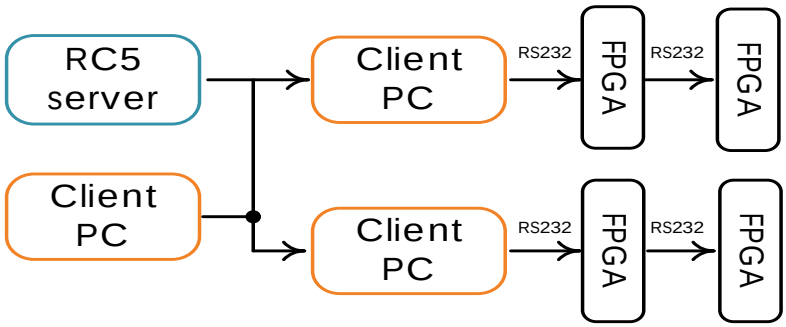


Figure 2.3: A hierarchical network where the FPGA hardware instances communicate with the server through the client PCs [6].

ent FPGA nodes on a TCP/IP network. The main idea to utilize *idle* FPGA resource areas by using the concept of *partial* runtime reconfiguration. For this purpose, a generic, scalable, high performance and platform-independent VHDL implementation of the hardware RC5 key searcher is developed in order to distribute it on various nodes in a client/server network. Figure 2.3 depicts the hierarchical setup of the network, which is organized on three different levels. On the highest level, the PC clients communicate with a centralized keyspace manager—termed as the RC5 Server. On the next level, the FPGA boards communicate with their corresponding PC clients through serial RS232 links. Finally, on the lowest level, the hardware key search instances communicate by using configured links inside FPGA. A client PC software is developed to deal with the communication between the PC and the FPGA board as well as between the Client PC and the RC5 Server. Each FPGA node can accommodate multiple instances of the key searcher on runtime, if sufficient area is available.

2.4.3 Bioinformatics Projects

Bioinformatics are compute-intensive applications suitable for distributed processing. Traditionally, these applications are addressed by cluster of workstations, which are normally general-purpose architectures not well-suited to meet the performance requirements. In recent years, high-performance FPGAs have emerged as a powerful paradigm for accelerating many bioinformatics search algorithms [37]. To further enhance their performance, many hybrid platforms are designed which integrate FPGA nodes

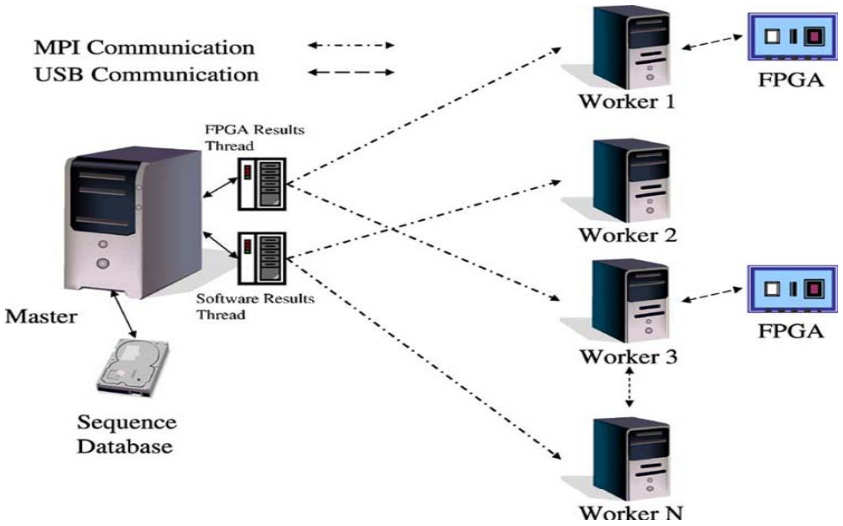


Figure 2.4: The MPI-HMMER-Boost project is a distributed system with FPGA nodes [7].

in a large-scale distributed system [64] [37] [65]. In the following, we briefly describe two different projects utilizing FPGAs in a distributed system.

The MPI-HMMER-Boost Project

The MPI-HMMER-Boost project [7] is a cluster-enabled hardware/software accelerated implementation of a famous bioinformatics database search tool—known as the *hmmersearch* [73]. The tool helps to construct important mathematical models of protein sequences, that can significantly reduce the search effort required to compare a particular sequence to a large database of proteins.

Figure 2.4 depicts the top-level design of the project. It is a combination of two acceleration techniques. The first technique—termed as the BioBoost hardware accelerator—makes use of FPGA-based implementation of the *hmmersearch* tool. Whereas the second is a software-based MPI [74] cluster implementation, called as the MPI-HMMER. The project combines these two techniques in the hopes to benefit from both technologies, using the MPI to scale beyond a single hardware accelerator. In Figure 2.4, the MPI-based cluster has been extended to include support for the FPGA accelerators.

Hybrid Computing Platform using Grid Network

In [37], an FPGA-based grid computing platform has been developed to accelerate the Smith-Waterman algorithm implementation for database search in a publicly available C code known as the EMBOSS suite [75], containing around 400 functions. In order to use the implementation in the platform, a Linux profiling tool gprof [76] was used to find out the most compute-intensive kernels to map them on an FPGA. To further enhance the performance of the platform, the FPGA-based hybrid computing platform was added as a client to a real grid environment by using the Globus Toolkit [21]. The total execution time for the implementation of C code in EMBOSS suite was considerably reduced, for different sizes of sequence database strings running on FPGA-based hybrid platform.

2.4.4 Reconfigurable Computing Clusters

Various projects have utilized FPGAs in the cluster computing environment to improve the performance of targeted applications, in terms of speed and scalability. A brief description of few such projects, is described in the following:

The first ever project to use independently operating FPGAs in a cluster system was **S**erverNet **E**nhanced **P**arallel **I**mage **A**ccelerator (SEPIA) [77]. SEPIA was developed as an image processing platform used to process 3D images. The PCI pamette FPGA-based boards were used to implement the network interface adapter to ensure the network communication between different cluster nodes.

Similarly, another project—termed as the adaptable computing cluster—made use of FPGAs in a cluster network by placing FPGA in the data path of the network [78]. The main focus was to improve the performance of a 2D FFT application.

The **R**econfigurable **C**omputing **C**luster (RCC) project aimed to analyze the possibility of designing a cluster of 64 nodes, to develop a Petascale computer [8]. The main purpose of petascale computing project is to test various computationally-intensive applications. Figure 2.5 depicts a block diagram of the cluster comprising of 64 FPGA-based compute nodes, a multi-Gigabit direct connect network and two Gigabit Ethernet switches. The Gigabit Ethernet network is responsible to provide TCP/IP access to cluster nodes and for distribution of FPGA configuration bitstreams to the nodes.

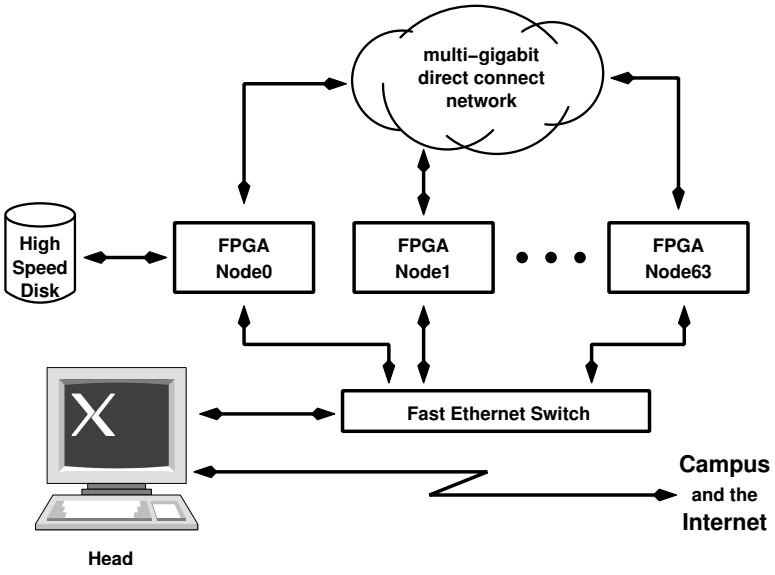


Figure 2.5: Block diagram of Reconfigurable Computing Cluster (RCC) [8].

Three different aspects of programming models and system softwares are discussed to develop the software infrastructure for the RCC project. First, a programming model to develop applications is required. Secondly, system level support to applications can be fulfilled by an operating system which is able to distributed jobs among the nodes. Thirdly, a mechanism to support bitstream management and distribution. The programming model adopted by the RCC based on MPI, as suggested in the adaptable computing cluster project [78] . The configuration bitstreams are remotely managed by using a system software known as *rboot* discussed in detail in [79]. The *rboot* system allows the students and the scientists to access a pool of 64 FPGA nodes over the Internet. The remote users are able to switch on/off the FPGA boards, upload/download files, configure the boards online and execute synthesized designs while sending input and output through the Internet.

2.5 Scheduling in Distributed Systems

Many large-scale scientific applications demand the coordinated processing of complex tasks, which require heterogeneous resources located within the realm of administrative domains. A typical scenario is the scheduling of com-

Schedulers	Host organization	Description
Grid Service Broker [81]	University of Melbourne, Australia	Scheduling system for computational and data grids
Nimrod/G [82]	Monash university, Australia	Resource manager and application scheduler, based on Globus toolkit
Condor/G [83]	University of Wisconsin, Madison, USA	Computation management agent for multi-institutional distributed systems
NetSolve [84]	The university of Tennessee, USA	A client/agent/server scheduling system

Table 2.5: List of important schedulers in distributed computing systems.

putational tasks in coordination with processors, storage, network and any other resources, such as software libraries, scientific instruments. A *scheduler* is a software that enables to coordinate the execution of this whole scenario. It should keep track of the required resources and create a complete schedule for the coordinated execution of complete workflow to run an application.

The proper utilization of resources in a distributed system largely depends on the coordinated task scheduling of applications to the computing nodes. There are several researches performed in the area of application task scheduling in distributed systems [41] [42] [43] [44] [80]. In their design, many factors are considered in terms of centralized vs. distributed, static vs. dynamic, and coarse-grained vs. fine-grained. In the following, we describe two important schedulers in grid systems. Table 2.5 enlists some important schedulers.

2.5.1 Nimrod/G

Nimrod/G is a generic resource scheduler, based on the Globus toolkit services and can be extended to any other middleware framework [82]. The design of the scheduler consists of three main components namely, the *task farming engine*, the *dispatcher*, and the *scheduler*. The task farming engine allows the users to create their own setup instead of using default environment. The dispatcher manages jobs by deploying the Nimrod/G agents on the computing resources. Finally, the scheduler performs the resource discovery and selection, and assigns the application tasks to proper resources in the system. All these component coordinate with each other to execute a

workflow environment. Nimrod/G is particularly useful for the scheduling of *embarrassingly parallel* applications, such as *task farming, drug design,* and *parameter sweep* applications [80].

2.5.2 NetSolve

NetSolve [84] is a client-server scheduling framework designed for the resource discovery and selection, and application job submission in the distributed computing environment. A user interacts with the server through a web interface in order to submit the applications. The server and client are linked through sockets, and the application is linked using specific APIs to the servers. The NetSolve server runs a daemon process to receive applications from the clients and executes them using various standard packages and libraries. The scheduler maintains the information about all the servers and clients and assigns jobs according to the current status and availability of servers.

2.6 Simulation Tools

In distributed computing paradigm, the performance analysis of different scheduling and resource management methods to get a desirable solution for a certain set of parameters is very important. This analysis is required to ascertain the feasibility of the design of a system before its actual deployment. Some important parameters to perform such analysis include the application makespan, the response time of a scheduling algorithm, the total tardiness, the waiting time of a job to get a service, the effect of number of nodes or processing elements, the network bandwidth, the availability of a particular resource, and the economics factors. Traditionally, the system designers and researchers have used three different methods for the performance analysis of a distributed system.

- **Theoretical Analysis:** Theoretical analysis is normally based on mathematical equations, which consider many unrealistic assumptions about the platform and the applications during the resource management and scheduling processes. Performance analysis of a computing system can become extremely complex when all the possible parameters and their impact are taken into account. Moreover, such analysis cannot be deterministic because of many factors, such as the probability of resource availability and its failure and dynamically chang-

ing bandwidth of networks etc. Due to these limitations of theoretical evaluation, most researchers obtain their results by either conducting experiments on simulators or on real testbeds, in order to get a more realistic analysis.

- **Real-world Platforms:** Second approach to obtain experimental results for performance analysis is to use real-world platforms, such as NorduGrid [59] or PlanetLab [85] etc. Using the web portals of the platform, a researcher can remotely log-in to the system, and use its powerful processing capabilities to obtain the required results. Such a method is more practical, but it is still undeterministic for getting results for experimental purposes, because most of such platforms are normally very busy in their own custom utilization. Furthermore, if a researcher wants to explore several different primitives to provide a sound statistical analysis for the experiments, it is important to conduct a significantly large number of experiments on real testbeds under a variety of load patterns. However, this process is time-consuming, and the results can be non-reproducible due to the dynamic nature of such platforms. Despite all these issues, the results taken from the real-world platforms are still considered as the most authoritative in research.

- **Simulators:** While the real-world platforms provide more realistic results to give an insight into the realization of a newly-designed computing platform or a scheduling heuristic, it is often very difficult to obtain all possible scenarios by conducting a large number of experiments on a real testbed. For this reason, researchers use simulators to verify the effectiveness of their designs. There are several distributed computing simulators used for the simulation experiments to study the various aspects in distributed systems. Some examples of these simulators are GridSim [45], SimGrid [48], OptorSim [86], GES [87], Bricks [88], and GangSim [89]. These simulators are capable of simulating computing resources, processing power of a certain resource, users, and network resources in a typical distributed computing environment. However, each of these simulators was developed to focus on a certain research area, such as grid economy, resource management, scheduling algorithm design and testing, accurate network simulation, and scalability. Moreover, these simulators are different from the network simulators, such as the NS2 [90], the OMNet ++ [91], the SSFNet [92], and the OPNET Modeler [93]. In case of the network sim-

Simulation tool	Website URL
GridSim [45]	http://www.buyya.com/gridsim/
SimGrid [48]	http://simgrid.gforge.inria.fr/
OptorSim [86]	http://edg-wp2.web.cern.ch/edg-wp2/optimization/optorsim.html
Bricks [88]	http://ninf.apgrid.org/bricks/index.shtml
GangSim [89]	http://people.cs.uchicago.edu/~cldumitr/GangSim/simu/v2.5.0sim/
GES [87]	Not available

Table 2.6: Website URLs of current simulation tools.

ulators, the primary focus is the analysis and design of communication networks, protocols, and related applications. They are originally designed for networking community, and they simulate networks on the packet level, providing the effect of network topology on the behavior of a network. Whereas, the grid simulators concentrate on the overall communication times and computing power of nodes with various resources. Similarly, they focus on the performance of computing resources, while the network simulators mainly concentrate on the network resources, such as the routers, the switches, and the network processors etc.

In a distributed system, computing resources are geographically dispersed in the form of computing nodes. When a user submits a job for processing, a scheduler assigns it to a particular node which is responsible to execute it. Once a job is processed, the output data is produced and sent back to the user. Normally, there are certain attributes associated with a computing resource. These attributes include resource ID, architecture, number of machines, number of processing elements (PEs), processing capability of each PE, cost, and the communication details. From the simulation point of view, these resource attributes are either represented in the form of mathematical equations which can be simulated, or by using discrete event simulations (d.e.s), where the operation of a system is represented as a chronological order of events and actions.

In the following sections, we briefly discuss various simulation tools in the distributed computing systems. Subsequently, we describe a comparison between these tools. Table 2.6 enlists the names and the website URLs of some important simulators. Apart from these simulation tools, there are some emulation tools developed to emulate a particular distributed system. For instance, MicroGrid [47] enables a controlled emulation of any system that is

developed by utilizing the Globus toolkit [21].

2.6.1 Bricks Performance Evaluation System

Bricks [88] is a performance evaluation system—developed in Java—to evaluate a global computing system by investigating the network behavior and scheduling systems. It uses client-server model, and makes use of a centralized scheduler to access scientific libraries and packages by remote servers requested by a client. The system operates as a discrete event simulation environment, where the network is modeled as a queuing system of servers and clients. The servers and network are represented in the form of queues, whereas the user tasks are viewed as number of instructions and the amount of data to be processed during the computation. The interaction between different servers is controlled by a centralized scheduling unit. New scheduling algorithms can be designed for Bricks system by changing the existing Bricks scheduling unit. The tasks are processed by a computing server in **F**irst **C**ome **F**irst **S**erved (FCFS) manner and it is modeled as a queuing system within a network of other servers. These tasks are specified as jobs entering into a queuing system at some specific arrival rate that is normally based on a real system.

2.6.2 OptorSim

OptorSim [86] is a Java-based simulator, originally developed for the European DataGrid project [94]. The basic purpose of the simulator is to provide various replica optimization strategies and testing different scheduling algorithms for a data grid. It includes **C**omputing **E**lements (CEs) and **S**torage **E**lements (SEs) assembled in the form of a grid site, to which, the user job and data are sent for the computation. A resource broker controls the job scheduling to grid sites, whereas a **R**eplica **M**anager (RM) handles the automatic replication of files on the grid sites using replication algorithm. The CEs and resource broker are simulated by utilizing Java threads in the simulator. The simulator can be configured using various configuration files that define the grid topology, resources, jobs, and scheduling algorithms. Once the simulator is configured and run, it provides different statistics, such as the total and individual job simulation times, number of replications, local and remote file accesses, the storage filled and percentage of time when CEs are working. The simulator can be run on command line as well as using a GUI and it provides a simple user guide.

2.6.3 Grid Economics Simulator (GES)

The main purpose of Grid Economics Simulator [87] is to study grid economy in the perspective of resource management. It offers an extendible and reusable framework for simulation of different economic resource management algorithms. For this purpose, the simulator includes base classes for grid entities, such as the consumers, the providers, the jobs, the resources and the environment. It is useful for the simulation environments, where the main focus is the scalability of a simulator for evaluating the effect of number of jobs or consumers during a simulation run. The simulator uses Java single-threaded discrete-time based simulations, and it is not sufficiently documented and is only limited to the grid economy research.

2.6.4 GangSim

GangSim [89] is focused on the study of usage policies in various grid sites and user groups, which are termed as the **V**irtual **O**rganizations (VOs). A *usage policy* is defined as the allocation of resources across many grid sites. The best solution is considered as the one, which determines the most optimal allocation of computing resources to a particular user group. For instance, a usage policy can be the allocation of 40 percent of the resources of a particular grid site to a VO, where each user group submits the jobs grouped together in the form of workloads. Each node in a grid site is modeled in terms of computing power, storage disk and network. Different configuration files are provided to describe the characteristics of the resources, network details, and usage policies. The usage policy defines the requirements—in form of the processing time, the disk storage, and network bandwidth—of a group of users (VOs) associated with a certain grid site.

2.6.5 SimGrid

The SimGrid [48] simulation toolkit has been developed at the university of California at San Diego. It was originally written in C language, but now it is available in both C and Java, and it is based on discrete event simulation. The main purpose of the simulator is to study various scheduling policies for the grid applications. The resources are modeled in time-shared fashion and can be described in terms of standard machine capability (MIPS). The application tasks are assumed as parts of a large application, and are defined in terms of their execution times. Each task is assigned to a resources in the grid, using

the user-defined scheduling algorithms. The network topology and links can also be defined, and the communication events are considered as independent tasks. A user can define a scheduling policy to assign application tasks on computing resources.

SimGrid has been utilized in a number of research works and real-world studies, and it is a very powerful simulation framework particularly useful in application prototyping. Earlier versions of SimGrid had centralized scheduling system, but it was later adopted to decentralized scheduling. One of the limitations of SimGrid is that it only provides *time-shared* resources. In this case, whenever a new job arrives, the system assigns it to one of the resources, and the overall execution time is updated for all the current jobs. Whereas, in many real-world situations, the resources are normally *space-shared* machines.This means that when a new job arrives, it is assigned to a PE which is idle, otherwise it is queued.

2.6.6 GridSim

Developed at Monash university, Melbourne, GridSim [45] is the most comprehensive simulation tool in scheduling and resource management in grid systems. It was initially designed to only study the economic grid resource management, but later, it was extended to perform research in various grid perspectives, such as grid scheduling policies, resource management, study of data grids etc. The simulator offers packet-level simulations of a network in which, the user entities can submit their jobs to the network through network routers. Moreover, it provides an output statistics framework, that generates useful statistics required for the study of scheduling algorithms. These statistics include the total simulation time, the effect of number of nodes, the resource economy etc. It supports both *space-shared* as well as the *time-shared* resource allocation. The clusters can be modeled as a single entity. GridSim also provides a customizable **G**rid **I**nformation **S**ervice (GIS), that is a useful module responsible for the registration, indexing and discovery of the resources. Furthermore, it supports the provision for data grids by providing the corresponding components, such as the *ReplicaCatalogue* (RC) which can be configured to implement user-defined replica algorithms. Every simulated entity in GridSim extends the GridSimCore class which includes both the input and the output objects to send and receive events. The toolkit provides the following important features:

- Heterogeneous types of resources can be modeled, where each re-

Simulator	Main features
GridSim [45]	New resource creation, grid economy, scheduling, network modeling.
SimGrid [48]	Application modeling, network analytical model.
OptorSim [86]	File replica management.
Bricks [88]	Client-server model, scheduling, scripting language for configuration.
GangSim [89]	Allocation policies for grid sites and Virtual Organizations (VOs).
GES [87]	Resource management, scheduling and scalability.

Table 2.7: Main features of the existing simulation tools.

source consists of a user-defined number of PEs.

- The computing power of each resource can be defined in the form of **M**illion **I**nstructions **P**er **S**econd (MIPS) according to the SPEC benchmark [95].
- It supports both the *space-shared* and the *time-shared* allocation policies for resources.
- Time zones can be defined for each resource.
- Advance reservation of resources can be modeled.
- Different parallel application models can be simulated.
- Any number of application jobs can be submitted to a resource.
- Multiple tasks from different users can be allocated to a single resource, in a *time-shared* or *space-shared* manner. This provides flexibility to test different scheduling algorithms.
- Both static and dynamic scheduling policies can be implemented.
- It provides an output statistics framework.

Every job from a grid user entity is submitted to a *broker entity*, which schedules it according to a well-defined scheduling algorithm. The broker entity is connected to the GIS entity, which keeps the lists of all the available resources in the system. Depending on the user-defined scheduling algorithm, the job is sent to the most appropriate resource accordingly. Broker entity plays an important role to define the economics in a grid network to satisfy the needs of all users and resource owners.

Simulation tool	Resource model	Task representation	Network	Application model
GridSim [45]	SPEC-based (in MIPS)	MIs	Packet level d.e.s	User defined
SimGrid [48]	SPEC-based (in MIPS)	MIs	math/d.e.s	SimDAG, MSG, SMPI
OptorSim [86]	Abstract amount	Instructions	Math	Abstract model
Bricks [88]	Abstract amount	Instructions	Parameterizable Client/Server model	Abstract model
GangSim [89]	–	Computation time	No network backbone	Abstract model
GES [87]	Abstract amount	Computation time	No network backbone	Abstract model

Table 2.8: A comparison—in terms of resource model, task representation, networking methodology, and application modeling—among different simulation tools in distributed computing research.

The GIS entity registers all the resources and maintains their updated statuses. The user jobs (in MIPS ratings) are submitted to the broker entity, which inter-operates with the GIS for the query of resources available according to the requirements of the user jobs. Once a suitable resource is found, the broker entity schedules the jobs according the scheduling policy defined by the user.

2.6.7 A Comparison between Simulation Tools

In this section, we compare these simulation tools by discussing their main features, resource modeling, application models, task representation, resource allocation policy, and network backbone. Table 2.7 enlists the main features of each simulation tool. Fundamentally, these simulators were designed for testing scheduling algorithms, grid economy and resource management policies. GridSim offers the flexibility to create new resource types by extending its resource class. Furthermore, it is also the only simulator that allows both the *time-shared* as well as *space-shared* resource allocation policy. In the *space-shared* policy, when a job arrives, it immediately starts

Simulation tool	Cost	Flexibility	Documentation	Language
GridSim [45]	Open source	New resources can be modeled New scheduling algorithms Time/space shared modeling	Fully available online	Java
SimGrid [48]	Open source	Applications modeling Network simulations	Fully available online	C and Java
OptorSim [86]	Open source	Scheduling and replica algorithms design	Insufficient	Java
Bricks [88]	Currently not available	CS network design for grid applications	Not available	Java
GangSim [89]	Available	Resource allocation policies for VOs	Insufficient	Java
GES [87]	Not available	Grid nodes scalability	Not available	Java

Table 2.9: The extensibility comparison among various simulation tools.

execution if a free PE is available, otherwise, it is queued. In the *time-shared* policy, an incoming job shares the resources (PEs) among other jobs in the system.

Table 2.8 gives a summary of resource modeling, task representation, network, and application models used in different simulators. In GridSim, the network is represented on packet-level discrete event simulation, where a user can submit jobs to a particular network resource, such as routers. The main focus of SimGrid is application modeling and network simulation, that is represented as a mathematical model of communication delays between resources.

Table 2.9 gives an extensibility comparison between the simulation tools, by focusing on the areas where a certain simulator is more flexible. For instance, GridSim is flexible to allow the modeling of new resources and new scheduling algorithms. Table 2.9 also provides the documentation details and the development language. Such a comparison can be useful in determining the possibility to utilize a certain simulation tool in a particular research perspective.

2.7 Incorporating Reconfigurable Nodes in Simulation Tools

In [49] [96] [97], GridSim simulator is extended to incorporate reconfigurable computing nodes by introducing a set of primitives, where some resources in the system contain **R**econfigurable **P**rocessing **E**lements (RPEs). The extended version of the simulator is termed as **C**ollaborative **R**econfigurable GridSim (CRGridSim), and it offers reconfigurable resources as hardware accelerators in the system. A certain compute-intensive application can exploit the availability of such accelerators in the system. However, the extensions in CRGridSim are limited, as they only include speedup factor of the accelerators over their GPP counterparts, as the only reconfiguration parameter. Apart from the speedup factor, the simulator introduced the reconfiguration delay when a certain RPE goes through the reconfiguration process. But it ignores several other important parameters, such as reconfiguration method, device area utilization, scheduling techniques, reconfigurability, hardware technology, and application model etc [49].

GridSim is an extensive simulation tool that allows to model many aspects—for example, resources, grid economy, scheduling techniques, and reservation strategies—of a distributed computing. However, it is primarily designed for GPP resources, and their computing capability is defined in terms of MIPs ratings. A resource can be modeled as a set of machines, each containing a number of GPP processing elements. Before conducting a simulation, these resources are predefined, and they have fixed computing power. Due to this reason, a resource specification cannot be modified to add reconfiguration parameters that require dynamic organization of resources. Additionally, the *resource information system* in GridSim is not capable of maintaining the dynamic requirements of reconfigurable nodes. Therefore, it was not feasible to make further extensions in the CRGridSim to add more reconfiguration parameters in the tool [50].

In the following, we briefly describe the parameters that must be taken into account, when developing a simulation tool for integrating reconfigurable nodes in distributed computing system.

2.7.1 Reconfigurable Area

It is the area—quantified in **L**ook-**U**p **T**ables (LUTs) or slices in an FPGA device—that can be utilized to reconfigure a device in order to execute a task.

To model and simulate a reconfigurable node, the information about its area must be taken into account. In a dynamic environment, the resource information system in a simulation tool should be updated according to the available area on the node.

2.7.2 Reconfiguration Delay

It is the time required to reconfigure a node. It is highly dependent on the size of bitstream of a configuration and the reconfiguration speed of the node. In a simulation tool, this delay can be modeled as a time elapsed to reconfigure the node, when a new task requires reconfiguration of the node.

2.7.3 Application Task

When a reconfigurable node is being set up for the execution of a task, it first requires the bitstream of the configuration required by the task, and then the application code and data are transferred to the node.

2.7.4 Reconfiguration Method

It is also important to consider whether the nodes in the system can execute multiple tasks simultaneously. In case of *partial* reconfiguration method, a node can be reconfigured for multiple configurations, and can process task(s) on each of them.

In [50] and [51], we introduce a simulation framework that incorporates reconfigurable nodes in distributed systems. It is a generic framework that takes into account all the above-mentioned factors. It can be utilized to test scheduling strategies and resource management aspects of reconfigurable nodes in distributed systems. The framework offers to model reconfigurable nodes, configurations, and application tasks. It offers to exploit several reconfiguration parameters to model the processor configurations, for instance, area required, and reconfiguration time delay etc. It allows to set up a simulation setup by adjusting many other parameters such as, node area, reconfiguration method, task arrival distribution, task arrival rate, reconfiguration delay range.

2.8 Summary and Conclusion

In this chapter, we introduced distributed computing systems and reconfigurable computing, and discussed various projects involving reconfigurable processors in their setup. We surveyed several middlewares, scheduling softwares, and simulation tools. Subsequently, we compared the various aspects of important simulation tools. We also argued why it is necessary to develop a new tool for the modeling and simulation of reconfigurable processor in distributed systems. Finally, we introduced the set of parameters required to develop a new simulation framework that incorporates reconfigurable nodes. In the next chapters, we give a detailed description of the design and implementation of our proposed simulation framework.

Note.

The content of this chapter is based on the following technical report:

M.F. Nadeem, F. Anjam, and S. Wong. **An Overview of Grid Softwares and Applications to exploit the added performance of Reconfigurable Hardwares in Grid Networks**. Technical report, Delft University of Technology, Netherlands, January 2009.

3

Virtualization of RPEs in Distributed Systems

In Chapter 2, we described that the traditional distributed computing systems are often termed as **V**irtual **O**rganizations (VOs) due to a seamless software layer—known as the *middleware*—between the application developers and hardware resources. These computing resources are currently the **G**eneral-**P**urpose **P**rocessors (GPPs). However, we expect that, in the design of next-generation distributed and high-performance computing systems, **R**econfigurable **P**rocessing **E**lements (RPEs) such as FPGAs and multi-core heterogeneous computers will play an important role. FPGAs are renowned for their power efficiency, programmability, performance, flexibility, and their scalability. Due to these characteristics, the recent and future computing systems are incorporating FPGAs as their core **P**rocessing **E**lements (PEs). Consequently, the design efforts to develop such heterogeneous computing systems are becoming complex, facing many challenges. One such challenge is the virtualization of resources in this new scenario. Many efforts to design proper virtualization schemes at higher software abstraction layers to utilize these RPEs are currently under research. In this chapter, we present a generic virtualization framework for distributed computing systems that supports RPEs. First, we present various scenarios in terms of use-cases to discuss the utilization of RPEs in distributed computing systems. Secondly, we propose a scheme to virtualize RPEs in distributed systems. Based on various virtualization levels, we provide a general model for a computing node which incorporates both GPPs and RPEs. Thirdly, we present a typical application task model. Finally, we present a case study of a large-scale application from the bioinformatics domain, which demands different types of processing elements in a distributed computing system.

3.1 The Concept

Distributed computing systems, such as grid networks utilize computing resources that are geographically distributed over the globe to perform computations for large-scale scientific applications that exceed the capabilities of clustered desktop computer or even a supercomputer [20] [98]. Generally, the overall performance of a distributed computing system greatly depends on the processing power of the employed computing resources and till now, the main processing elements in these systems were (programmable) general-purpose (multi-/many-) core processors. However, due to growing demands of new scientific applications, more performance and power efficiency are required from the processing elements [55]. Therefore, new possibilities are opening up in order to utilize Reconfigurable Processing Elements (RPEs) such as FPGAs, in distributed systems [7] [8] [37] [39].

In recent decades, FPGAs have obtained growing attention due to their flexibility, power efficiency, and ease of use [28]. Some of the characteristics of reconfigurable hardware include, functional flexibility, power efficiency, hardware abstraction, ease of use, adaptability and short design time, extensible (adding new functionality), reasonably high performance, and scalability by adding more cores. Most significantly, the reconfigurable architectures are utilized as hardware accelerators in order to increase performance. In addition, the (re-)configurability of these architectures means that they can be optimized for different applications without overhead operations that are required when using traditional architectures result in less resource waste and reduced energy consumption. Furthermore, they also provide the programmability of a general-purpose processor.

However, the design complexity and utilization of the RPEs in a distributed computing system lead to innovative research and open problems. One such problem in distributed computing is the virtualization of resources by hiding the details of the underlying hardware. For the execution of an application task, it must be assigned to an instance of a computing resource. Oftentimes, a particular resource is shared by many tasks simultaneously, that can create an imbalance between the tasks and the available resources. Such an imbalance can be avoided by using proper task scheduling schemes and developing precise virtualization techniques for the resources. Virtualization permits a seamless coordination between the resource providers and users, in terms of resource sharing and solving scientific problems in a dynamic environment. This kind of sharing must be secure, coordinated, reliable, fault-tolerant, manageable, and highly controlled. All these characteristics are nor-

mally provided by a software layer—known as the *Middleware*—between the hardware resources and the users. For instance, the most famous middleware is the Globus toolkit, which is an underlying technology for designing distributed grid systems that aggregate the hardware resources to allow computing power, scientific tools, storage disks, and other instruments to be shared securely across various organizations and institutions [21].

Currently, the traditional distributed computing systems are already virtualized for GPPs; hence, they are often termed as *Virtual Organizations (VOs),* and there is a *hardware independent layer* between the tasks and the resources [98]. Although many attempts have been made to virtualize reconfigurable hardware in order to utilize them effectively for more than one application tasks [99] [100] [101], but with many limitations. Their main focus is on a single device [100] [102] or a multi-FPGA High Performance Reconfigurable Computing (HPRC) system, and none of them discusses the virtualization for RPEs in a distributed computing system.

In this chapter, we propose a general framework to virtualize a computing node in a distributed computing system, which contains GPPs as well as RPEs. The proposed virtualization framework can be used to obtain and demonstrate the following objectives:

- More performance can be achieved by utilizing reconfigurable hardware, at lower power.
- Due to abstraction at a higher level, an application program can be directly mapped to any of the RPE or the GPP.
- Reconfigurable hardware is expected to support reconfigurability and different hardware implementations on the same RPE are possible due to reconfigurable nature of the fabric.
- The resources can be utilized in a more effective manner when the processing elements are both GPPs and RPEs. Those distributed computing applications which contain more parallelism can get more benefit if executed on the reconfigurable hardware.

The proposed virtualization framework is adaptive in adding/removing resources at runtime. Application tasks can be seamlessly submitted to the distributed computing system. The main contributions of the chapter are the following:

1. We present and describe a set of various use-case scenarios of applications that can benefit from our presented virtualization framework.
2. Based on the use-case scenarios and their virtualization levels, we provide a generic model for a distributed computing node which integrates both general-purpose and reconfigurable processing elements. Similarly, we present a typical task model representing an application task which requires a specific processing element for its execution.
3. Finally, we present a specific case study of a real world large-scale application that takes benefit of the proposed framework, and utilizes various processing elements in a distributed system.

The remainder of the chapter is organized as follows. Section 3.2 presents some background on the virtualization of hardware resources. Section 3.3 presents possible use-case scenarios of applications utilizing various types of PEs. It also discusses virtualization/abstraction levels on distributed network with RPEs. In Section 3.4, we explain our proposed virtualization framework. In Section 3.5, we present a simple case-study from bioinformatics application domain, to show the utilization of RPEs in a distributed computing system. Finally, Section 3.6 describes the summary of this chapter.

3.2 Virtualization of RPEs — Background

This section provides the details of some recent attempts to virtualize the reconfigurable hardware to enable the execution of various application tasks. In a survey [99], the authors identified three different approaches of hardware virtualization. In the first approach called *temporal partitioning*, a large-scale application is divided into smaller parts which can be mapped onto full FPGA. Then, each part is run sequentially, until the whole application is executed. In the second approach called *virtualized execution*, a certain level of device-independence — for a particular family of FPGAs — is achieved through a proposed programming model that defines an application task as an atomic unit of computation. In this way, a runtime system is developed to ensure that the application can be executed on any FPGA which is part of a device family. In the third approach, a hardware *virtual machine* is proposed which can execute an application on a general abstract FPGA architecture. This provides an even higher abstraction level and is useful for those reconfigurable systems which can be utilized in a network of FPGAs. Similarly, [103] specified general methodologies for the virtualization of reconfig-

urable devices in hardware and software systems. The authors adopted the concepts used in the operating systems, such as *partitioning* and *overlaying*.

In [100], two different hardware resource virtualization techniques have been proposed, for dynamically partially reconfigurable systems. In the first approach — *logic virtualization technique* — an FPGA is configured for a set of application tasks at runtime. It is a many-to-one mapping which supports many applications, dynamically. In the second approach — *hardware device virtualization* — an FPGA is configured with more than one hardware functions. In this one-to-many mapping, a single software application can utilize many functions on a single device. Similarly, [102] proposed a virtualization layer for dynamically allocating hardware functions on a reconfigurable system, to execute any software application. However, the techniques presented in these works are focused on a single FPGA device.

In [104], the authors proposed an approach to virtualize and share reconfigurable resources in High-Performance Reconfigurable Computers (HPRCs) that contain multi-node FPGAs. The main idea is to utilize the concept of *virtual* FPGA (VFPGA) by splitting the FPGA into smaller regions and executing different task functions on each region. The validation process is carried out by using an HPRC system, Cray XD1, and implementing a layer of OS virtualization to manage the virtual regions. Although, this approach utilizes a multi-FPGA system, but it does not focus on a distributed system containing FPGAs.

All the above-mentioned works attempt the FPGA virtualization for a single node or a multi-FPGA system. However, in [101], an abstract model of *hardware virtual machine* is proposed for the networked reconfiguration of FPGAs. The main focus of the work is to propose a design flow model by identifying the responsibilities of a *client* and a *service provider*. In this model, both the client and the service provider become aware of the FPGA mapping tools — which they are required to maintain — in order to reconfigure the device and execute the application tasks. But in their approach, it is not discussed, how to manage the FPGA resources in the resource management system.

Various different *Workflow Management Systems* in grid computing to match tasks with different computing resources, have been surveyed in [105]. Most important of such systems is the Condor project [83]. The matchmaking modules in Condor, utilize the user-specified algorithms that match jobs with the available resources based on a defined criteria, such as classified advertisement [106]. It supports a mix of various GPP resources and matches

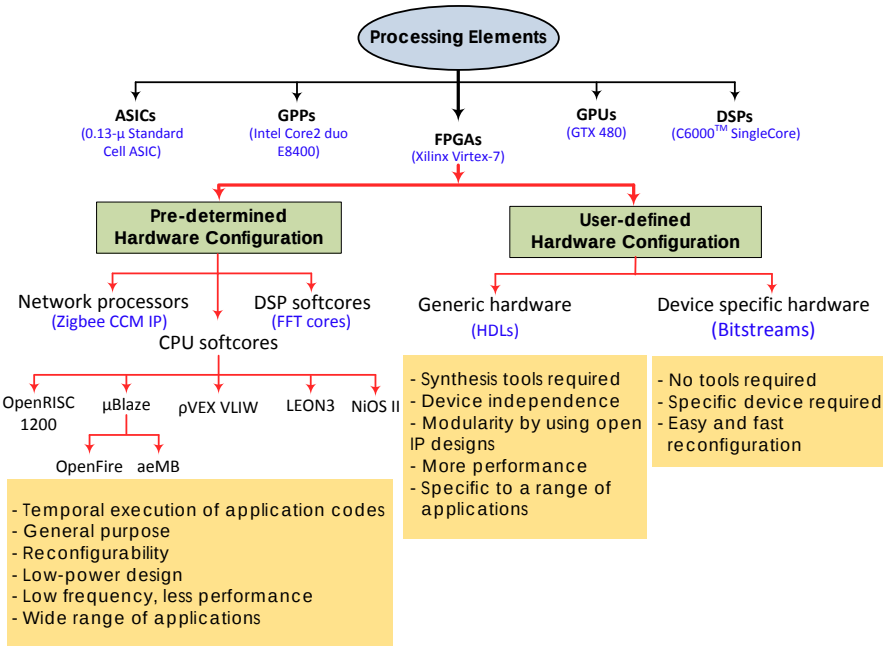


Figure 3.1: A taxonomy of enhanced processing elements.

computing jobs accordingly. However, it does not integrate the reconfigurable computing resources in such systems. It is expected that the future distributed systems will contain RPEs as their main processing elements, along with the GPPs. Therefore, the grid managers seek new methods for the virtualization of RPEs along with existing computing resources.

3.3 Different Use-case Scenarios

In this section, we discuss all possible use-case scenarios of applications that utilize RPEs. Each of these scenarios has certain characteristics that need to be translated to specific requirements. Therefore, a generic virtualization framework is required that includes provisions to map the requirements of all possible scenarios to the required RPEs. Figure 3.1 depicts a taxonomy of enhanced processing elements in high-performance domain. Such a framework should be flexible to virtualize a generic computing node in a distributed system, containing GPPs as well as RPEs. However, it should be extendable to add more types of PEs. Table 3.1 provides some examples of possible PEs. It

PE	Parameter	Description
FPGA	Logic cells, Slices, LUTs, Gates, Macrocells, ALMs	Designed to implement user-defined combinatorial and sequential functions.
	BRAM, Memory Blocks, Embedded Memory	Additional memory blocks available in terms of distributed RAM.
	DSP Slices	Pre-configured multiplier, adder, and accumulator required for high-speed filtering.
	Speed Grades	Maximum frequency at which a device can operate.
	Reconfiguration Bandwidth	Speed (in MB/s) to reconfigure a device
	IOBs	Support different I/O Standards
	Ethernet MAC	Embedded MAC for Ethernet applications
GPP	CPU Type/Model	Type of CPU
	MIPS ratings	Million Instructions per Second processing capability
	OS	Operating System
	RAM	Main Memory
	Cores	Total number of cores
Softcores (VLIW)	FU Type	Multipliers, ALUs
	Issue Width	Number of Issues
	Memory	Instruction and Data memory
	Register File	Register file size
	Pipeline	Number and Size of Pipelines
	Clusters	Number of Clusters
GPU	Model	GPU Model
	Shader Cores	Number of Data Parallel cores
	Warp Size	Number of SIMD threads grouped together
	SIMD Pipeline Width	Size of SIMD Pipeline
	Shared Memory/Core	Shared Memory per Core
	Memory frequency	Maximum clock rate of memory

Table 3.1: Parameters of different Processing Elements (PE).

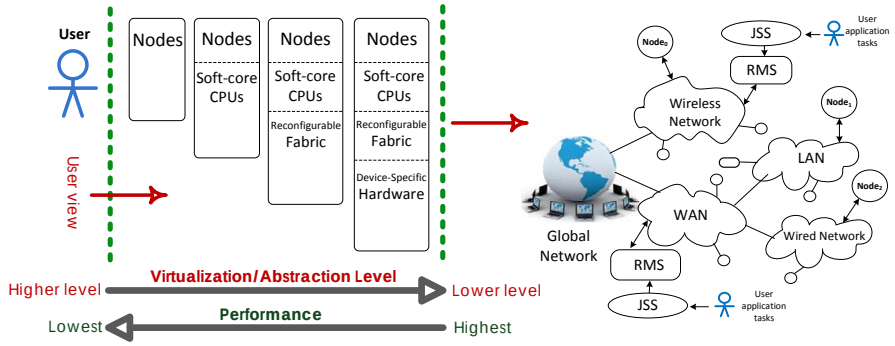


Figure 3.2: Different virtualization/abstraction levels on a reconfigurable distributed computing system.

is evident that the design complexity of such a generic framework increases enormously, due to dealing with various sets of parameters in each PE. Additionally, there are several different types and families within each PE, further increasing the complexity. Here, we only focus on RPEs, and discuss only relevant scenarios in this respect. In Section 3.3.1, we discuss a scenario with already existing software-only applications, whereas in Section 3.3.2, we describe hybrid applications which can be executed on either GPPs or RPEs.

3.3.1 Software-only Applications

Many existing applications are already developed for GPP-based distributed computing networks. Therefore, there is a need to provide mechanism on the next-generation distributed computing systems to provide *backward compatibility* and support such applications. Their performance on the new distributed computing network should be *similar if not better*. Therefore, this use-case scenario covers all the software-only applications already existing for the distributed computing networks. These applications are executed on the GPP nodes and are not aware of the reconfigurable fabric in the distributed computing system. However, if the distributed computing system can not provide the required GPP node to an application at some instance, it should be able to configure a soft-core CPU on a currently available RPE to obtain *similar if not better* performance. This use-case scenario is depicted in Figure 3.1 under the *pre-determined hardware configuration*.

3.3.2 Hybrid Applications

In this scenario, the application is aware of the reconfigurable fabric on the distributed computing network. Therefore, the application makes use of such computing nodes that contain both GPPs and RPEs. In this way, the performance of the application can be improved. Figure 3.2 depicts different virtualization/abstraction levels for a user. In this scenario, a user can view soft-core CPUs, along with the computing nodes. This use-case scenario can be divided into further parts, explained as follows:

3.3.2.1 Pre-determined Hardware Configuration

There could be a situation where some of the *compute-intensive* tasks in a certain application, are implemented in an optimized way to improve the performance. These optimizations for speed are specific to some particular architecture, for example, software kernels (FFTs, filters, multipliers etc.) optimized for VLIW, RISC, μBLAZE, etc. Therefore, these tasks need to be executed on those particular architectures. One example of such architecture is a soft-core ρ-VEX VLIW processor (P_{type}) implemented on an FPGA [107]. Depending upon the requirements of an application, it can be adopted to several parameters such as, the number of issue slots, cluster cores, the number and types of functional units, or the number of memory units.

Such applications consist of generic and the user-selected soft-core specific tasks. The generic tasks are mapped onto the GPPs, whereas the other specialized tasks are executed on the corresponding user-defined soft-cores. In this scenario (see Figure 3.1), the architecture becomes more general purpose in nature, allowing temporal execution of wide range of applications. It is normally low-power and low-frequency design which is more flexible, but with less performance.

3.3.2.2 User-defined Hardware Configuration

Open-source (e.g., the OpenCores IPs [108]) hardware are already available for specific tasks in many applications. Therefore, these designs — which are available in generic HDLs — can be reused, while developing the system applications. As depicted in Figure 3.1, this use-case scenario represents applications which are complex, and cycle (or performance) and/or data hungry. Hence, they consume a lot of time to finish their processing and provide results. For such applications, the performance can be significantly improved

by application-specific accelerators which can greatly reduce the time overhead and therefore, an application designer would like to provide such hardware accelerator to be configured on the fabric in the distributed computing system. Here, the application developer provides the hardware accelerator specifications in generic hardware description languages, such as VHDL and Verilog. On one hand, this scenario creates opportunities for the system managers to map the generic hardware accelerators on any of the available reconfigurable fabric in the distributed computing system to improve the utilization of its hardware resources. On the other hand, it also provides important *services*, such as mechanism and tools to generate device specific bitstreams for the user. In this use-case, the *service provider* is required to possess the synthesis CAD tools. A certain degree of device independence is achieved due to generic nature of user application specifications. However, the design is limited to a specific range of applications. On this virtualization level (see Figure 3.2), the *reconfigurable fabric* is visible to the user.

3.3.2.3 Device-specific Hardware

This is the lowest level of virtualization (see Figure 3.2) where the user is aware of the device specifications available in the distributed computing system. In this case, the user wants to make use of his own hardware design or IP for a particular device. Therefore, the distributed computing system should provide opportunity for such user to submit his/her applications. This scenario is suitable for the users with applications which require higher performance. In this case, the user can make use of a hardware of his/her own choice. The hardware is directly visible to the user. The cost of the high performance is long application development time. Because, the developer needs to provide a fully tested and synthesized bitstream of the design. In this scenario, the *service providers* are not required to possess the CAD tools. However, they are expected to provide the specific device targeted by the application developer.

3.3.3 Different Virtualization/Abstraction Levels

All these use-case scenarios lead to different virtualization/abstraction levels in a distributed computing system, as depicted in Figure 3.2. It can be noted that, as we go to a lower abstraction level, the user should add more specifications along with his/her tasks and get more performance, and vice versa. For instance, at the lowest abstraction level, a user must provide the details

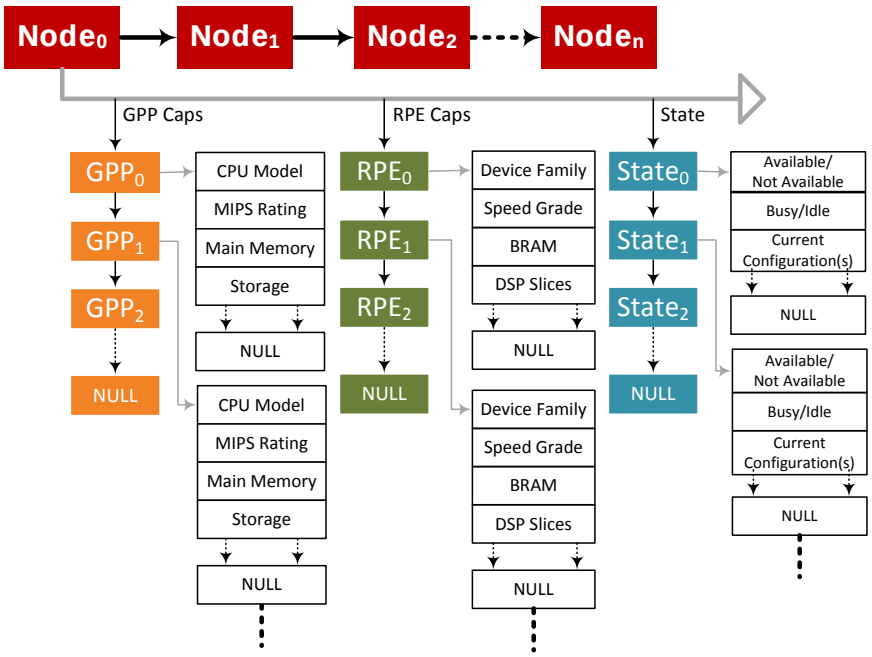


Figure 3.3: A typical computing node to virtualize RPE.

of device specific hardware along with the bitstream of their design and the application task data. At this level, the hardware is visible to the user, and the managers provide the access to the required device.

3.4 The Proposed Virtualization Framework

Based on the use-case scenarios discussed in Section 3.3, we describe our proposed framework for reconfigurable hardware virtualization in distributed systems. We propose models for a general distributed computing node and a generic application task which can integrate both GPPs and RPEs in a distributed computing network.

3.4.1 A Typical Node Model

Figure 3.3 depicts our proposed distributed computing node model to incorporate RPEs, along with GPPs. It contains a list of all processing elements (GPPs and RPEs) and their attribute, and can be defined as follows:

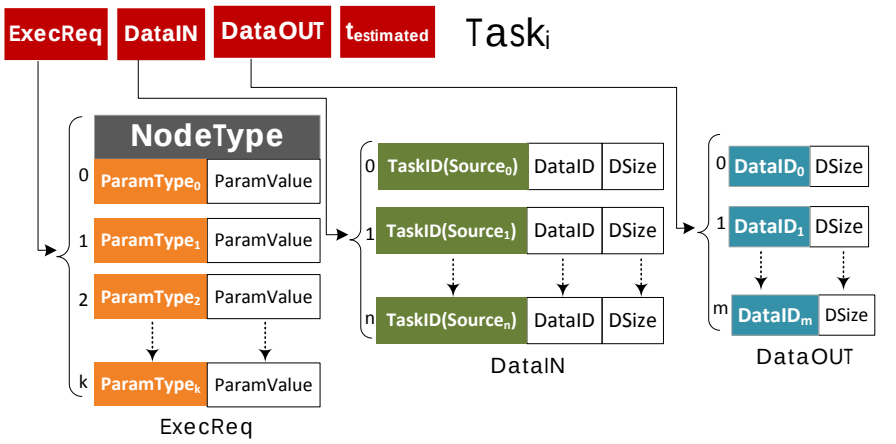


Figure 3.4: Application task virtualization for distributed computing system with RPEs.

$$Node\left(NodeID, GPP\ Caps, RPE\ Caps, state\right) \tag{3.1}$$

A typical distributed computing node contains a list of resources as depicted in Figure 3.3. Each resource consists of a null terminated list of GPPs, RPEs, and their current *state*. Each data member in the list of GPPs (or RPEs) is characterized by *GPP Caps* (or *RPE Caps*), which represents a set of parameters.

These parameters provide information about the capabilities of the GPP (or RPE). A typical list of these parameters is given in Table 3.1. Similarly, *state* represents the current states of different elements. It is a dynamically changing attribute of the node. For instance, the *state* can provide the current available reconfigurable area or maintains the information of current configuration(s) on an RPE.

The proposed node model is generic and adaptive in adding/removing resources at runtime. Furthermore, a resource information manager can add more parameter specifications of a particular processing element.

3.4.2 A Typical Application Task Model

Figure 3.4 depicts the proposed model for a typical application task. It is represented by the following tuple:

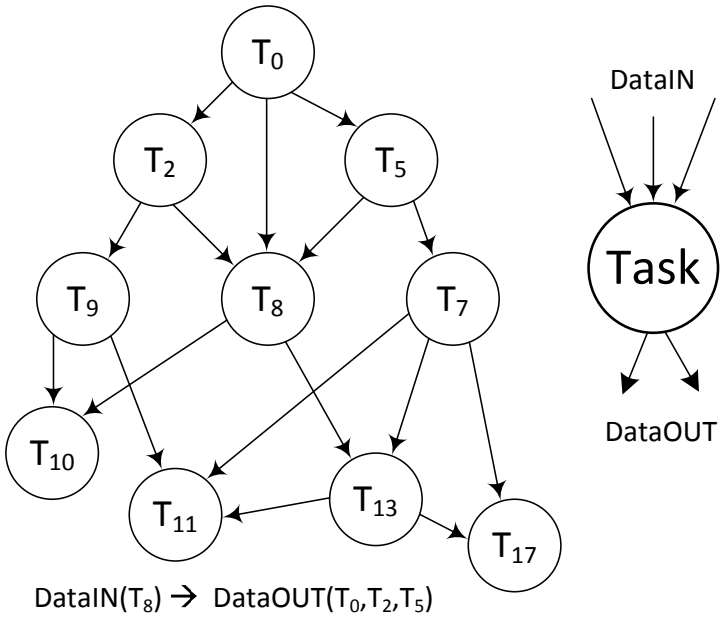


Figure 3.5: An application task graph.

$$Task\left(TaskID, Data_{in}, Data_{out}, ExecReq, t_{estimated}\right) \tag{3.2}$$

In tuple (3.2), TaskID provides the ID of a task. $Data_{in}$ and $Data_{out}$ identify the input and output parameters of the task. Whereas, the *ExecReq* gives the execution requirements of the task. $Data_{in}$ is completely identified by the ID of the source task (TaskID), DataID, and data size (DSize). Similarly, $Data_{out}$ provides the information about the output generated by the current task and again, it is identified by the DataID and DSize. TaskID, $Data_{in}$ and $Data_{out}$ provide enough information to the scheduler in a distributed computing system to assign this task to a node.

ExecReq provides the list of resources required by the task for its execution. This list is composed of the node type and its parameters. Each parameter is followed by its value. These parameters completely identify the architectural requirements by the current task. Finally, $t_{estimated}$ is the estimated time for completion of this task, when it is executed on a particular processing element, identified by its *ExecReq*.

A typical task is illustrated by Figure 3.4. The input data is represented by DataIN and it can be initiated by n number of sources. Similarly, a task can

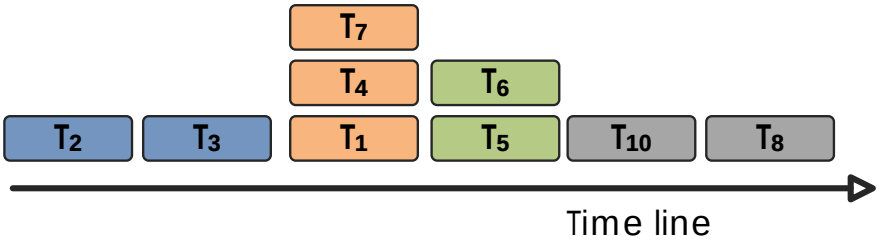


Figure 3.6: An example of application tasks execution given in tuple 3.4.

produce m number of different outputs represented by DataID and DSize. Finally, the *ExecReq* provides list of k parameters which define a typical *NodeType* required to execute the task. The data dependencies among different tasks are represented by an application task graph in Figure 3.5. From example, it can be noticed that inputs to T_8 are the outputs of tasks T_0, T_2, and T_5. Similarly, $DataIN(T_{11}) \rightarrow DataOUT(T_7, T_9, T_{13})$, $DataIN(T_{13}) \rightarrow DataOUT(T_7, T_8)$, and $DataIN(T_{17}) \rightarrow DataOUT(T_7, T_{13})$.

The user provides its application in form of tasks and their dependencies. A typical application is represented by the following tuple.

$$Application_i \left(< Keyword >, Task\ list, < Keyword > \right) \tag{3.3}$$

Each application is identified by a keyword followed by a task list; whereas, a keyword shows whether the tasks can be executed in series or parallel. For example, a particular application is given as follows:

$$App\{Seq(T_2, T_3), Par(T_1, T_4, T_7), Par(T_5, T_6), Seq(T_{10}, T_8)\} \tag{3.4}$$

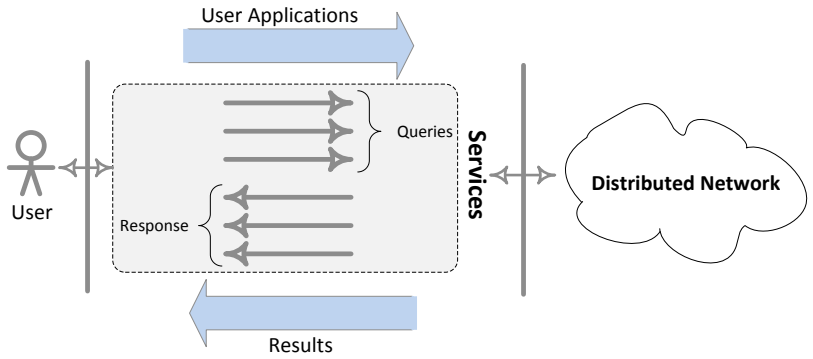


Figure 3.7: User *services* in a typical distributed computing system.

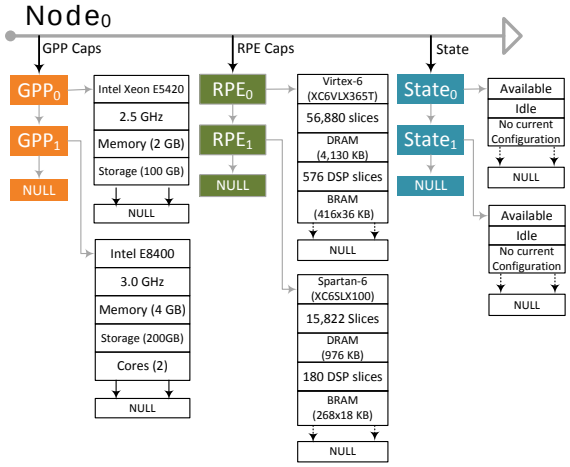
In this example, the tasks T_2 and T_3 should be executed sequentially, along with the parallel execution of the task lists (T_1, T_4, T_7) and (T_5, T_6), followed by the sequential execution of the task list T_{10} and T_8. The keywords *Seq* and *Par* show that the task lists followed by these keywords should be executed sequentially or in parallel manner, respectively. Each task list is terminated by next keyword. The sequence of task execution in this example is illustrated in Figure 3.6.

Figure 3.7 depicts different user *service* levels in a distributed computing system. The minimum level of services required by a user is to submit his application tasks and get results. But more services can be added to satisfy the Quality of Service (QoS) requirements. These services include cost, monitoring, and other user constraints. With these services, a user is able to submit his/her queries and get a response as depicted in Figure 3.7.

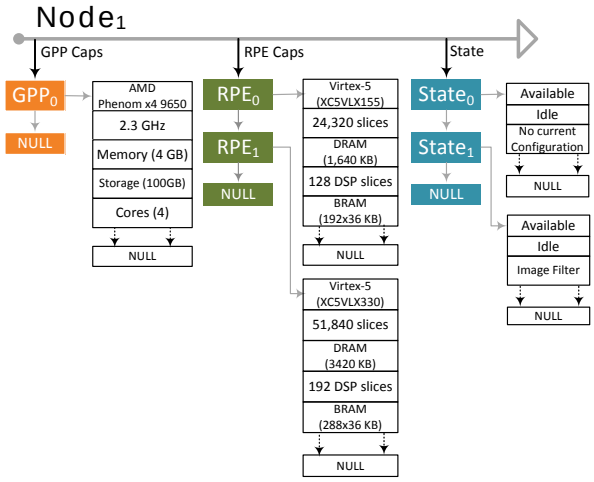
3.5 A Case-Study from Bioinformatics Application Domain

In this section, we describe a generic case-study to provide the concept of virtualization of reconfigurable elements in a distributed computing system. We give an example of a simple application and a distributed computing network consisting of 3 different nodes. Furthermore, we elaborate on the role of resource management system and application task scheduling.

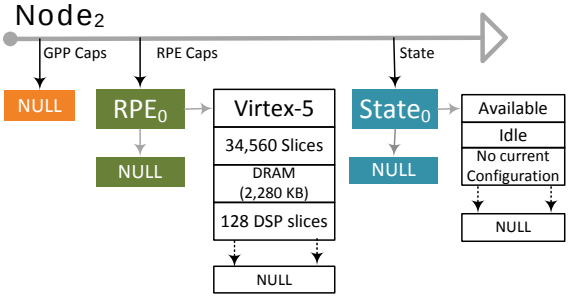
In Figure 3.2, a general view of a distributed computing network is illustrated. It contains different computing resources in the form of nodes. Each node is characterized by different parameters as depicted in Figure 3.3. The



(a) $Node_0$.

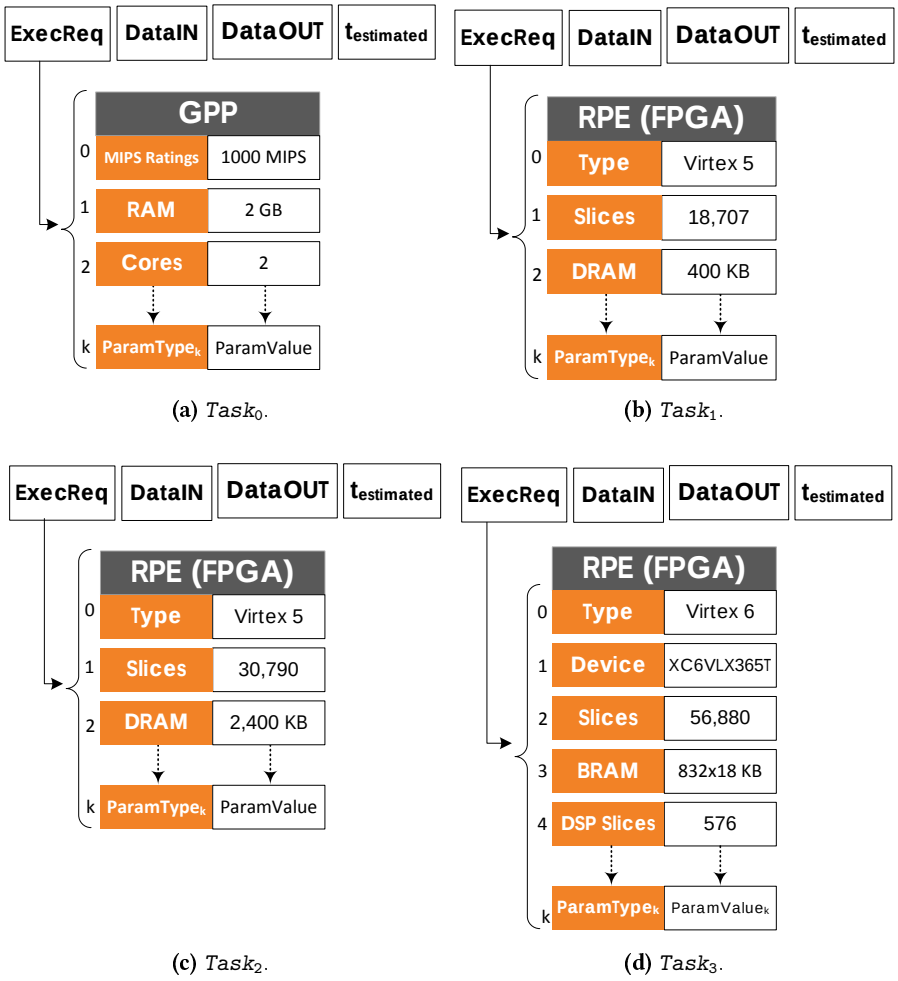


(b) $Node_1$.



(c) $Node_2$.

Figure 3.8: Specifications of three computing nodes in the case-study.



(a) $Task_0$. (b) $Task_1$.

(c) $Task_2$. (d) $Task_3$.

Figure 3.9: Execution requirements for the specifications of four tasks in the case-study.

distributed computing network contains various *Resource Management Systems* (RMS) along with the *Job Submission System* (JSS). A user submits his application tasks through a JSS. Each application task is part of a large application and it is depicted by Figure 3.4. The RMS updates the statuses of all nodes in the system. It also implements a task scheduler which assigns the user application tasks to different nodes in the network. The scheduling decisions are governed by a task scheduling algorithm and the availability of nodes.

For our case-study, we consider a distributed computing network which contains 3 different nodes, denoted by $\texttt{Node}_0$, $\texttt{Node}_1$, and $\texttt{Node}_2$, as depicted by Figure 3.2. Figures 3.8a, 3.8b, and 3.8c depict the specification of these nodes. It can be noticed that $\texttt{Node}_0$ contains 2 *GPPs* and 2 *RPEs*. The $\texttt{State}_0$ and $\texttt{State}_1$ provide information that both *RPEs* are currently *available* and *idle*. Moreover, they are not configured with any processor configuration. Each GPP and RPE is defined by a list of parameters which completely provide its characteristics. Similarly, $\texttt{Node}_1$ contains one *GPP* and 2 *RPEs*, and $\texttt{Node}_2$ consists of only one *RPE*.

We consider a large-scale application in the bioinformatics domain, from a famous benchmark suite, known as BioBench [109]. We choose and analyze ClustalW from the suite, which is a representative multiple-sequence alignment application. It enables the process of finding similarity between more than two DNA sequences [110]. For an analysis of ClustalW, we first identified *compute-intensive* methods in the application using gprof [76], which is a GNU software tool for profiling. Figure 3.10 depicts gprof profiling graph of the top 10 most compute-intensive kernels in the ClustalW application. Secondly, for possible mapping of ClustalW application on an *RPE*, we used a tool a quantitative prediction model for hardware/software partitioning, called Quipu [111]. It is a linear model based on software complexity metrics (SCMs), and can estimate the number of slices, memory units, and look-up tables (LUTs) within reasonable bounds in an early design stage. Furthermore, such a model can make predictions in a relatively short time, as required in a hardware/software partitioning context. We identified that two main functions *pairalign* and *malign* contribute to the 89.76% and 7.79% of the total time consumption of the application. Using Quipu tool, we estimated that *pairalign* requires 30,790 slices, whereas, malign requires 18707 slices on Virtex 5 devices.

Based on the analysis using profiling information, we consider that the ClustalW application can be divided into three generic tasks, as depicted in

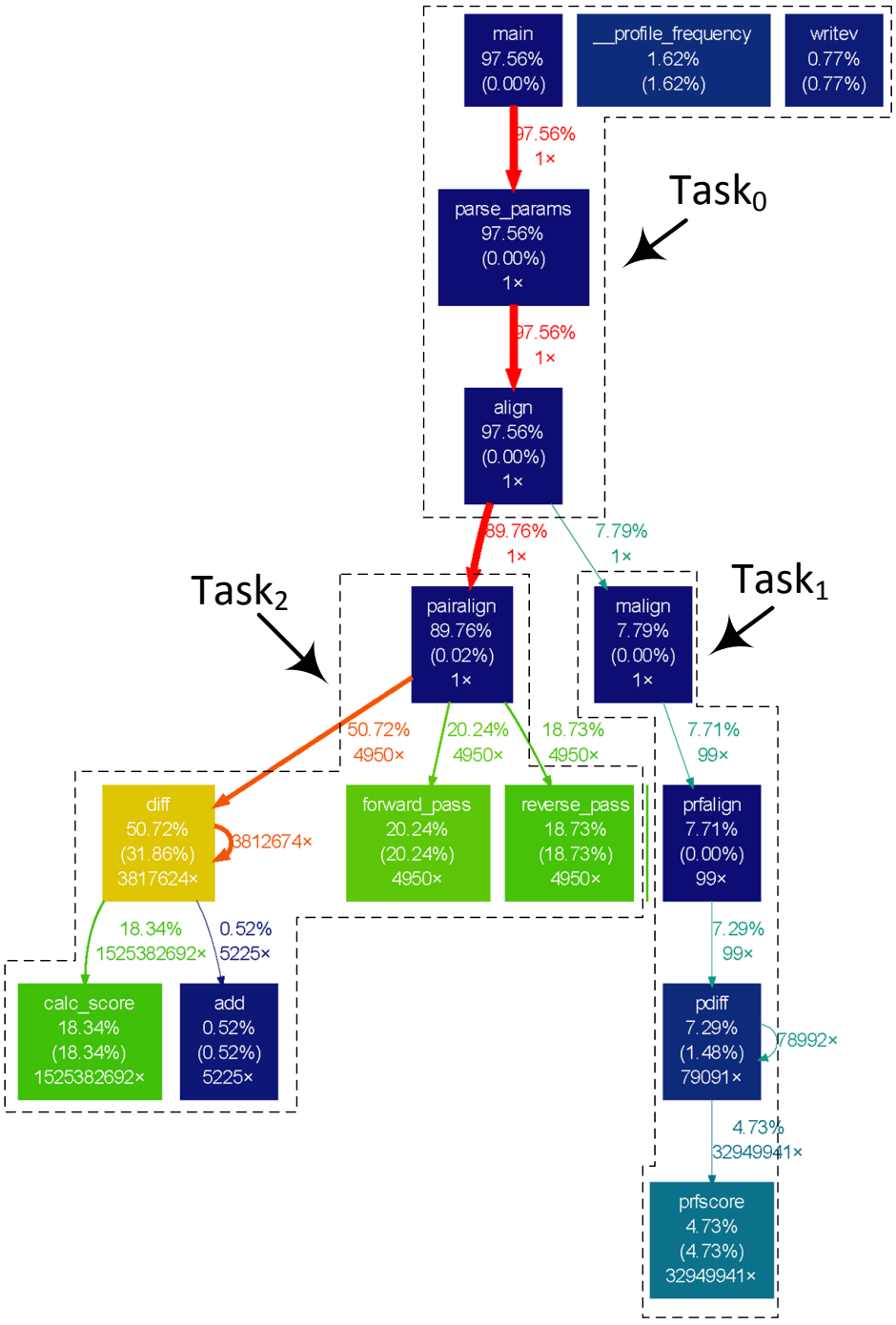


Figure 3.10: Time profiling of the top 10 *compute-intensive* kernels in the ClustalW (BioBench) benchmark using gprof tool.

Figure 3.10. The execution requirements *(ExecReq)* of each task are depicted in Figures 3.9a, 3.9b, and 3.9c. Similarly, we consider that an application developer implements a device-specific hardware of the whole ClustalW application as one hardware task and submits its *ExecReq* which are represented by Figure 3.9d. These tasks are submitted to a certain JSS which analyzes the requirements of each task and forwards it to the RMS. The RMS implements a task scheduler that takes decisions to assign each task onto a particular node specified by some task scheduling algorithm. In our case study, we analyze the mapping options for each task (given in Table 3.2) in the following:

$\texttt{Task}_0$: Since the profiling information in Figure 3.10 shows that this task only distributes data to the *malign* and *pairalign* functions, so it can be considered as a task requiring a GPP only. It can be noticed that any of the GPP_0 and GPP_1 in the $Node_0$ and GPP_0 in the $Node_1$ contain the minimum processing requirements by the $Task_0$. Consequently, the scheduler can assign it to any of $Node_0$ or $Node_1$. The user provides the application code and required input data. $Task_0$ is a typical example of use-case scenario mentioned in Section 3.3.1.

$\texttt{Task}_1$**:** requires a Virtex-5 FPGA device with minimum of 18,707 slices. It is evident from Figure 3.8 that RPE_0 and RPE_1 in $Node_1$ and RPE_0 in $Node_2$ all contain Virtex-5 type devices with more than 24,000 slices, so the $Task_1$ can be assigned to any of $Node_1$ or $Node_2$. The user provides the configuration details (HDL specifications or bitstream), along with application code and input data, to the RMS. $Task_1$ is an example of use-case scenarios given in Sections 3.3.2.2 and 3.3.2.3.

$\texttt{Task}_2$**:** is similar to $Task_1$ and requires at least 30,790 Virtex-5 slices. These requirements can only be met by the RPE_1 in the $Node_1$ and RPE_0 in the $Node_2$. Similar to $Task_1$, it also represents use-case scenarios given Sections 3.3.2.2 and 3.3.2.3.

$\texttt{Task}_3$**:** in this scenario, the task requires a particular device-specific hardware (Virtex XC6VLX365T). The application developer designs and implements its design on a particular targeted hardware device and provides its bitstream. The user submits the application along with the specific bitstream and data. In this case, $Task_3$ can be assigned to RPE_0 in the $Node_0$ only.

All possible mapping options and user-selected abstraction levels are given in Table 3.2. The mapping decisions are based on a particular scheduling strategy implemented inside the scheduler in the RMS, that takes into account various parameters, such as area slices, reconfiguration delays, and the time required to send configuration bitstreams, the availability and cur-

Task	Possible mappings	User-selected abstraction levels
$Task_0$	$GPP_0 \leftrightarrow Node_0$, $GPP_1 \leftrightarrow Node_0$, $GPP_0 \leftrightarrow Node_1$.	*Software-only application* **OR** *Predetermined hardware configuration*
$Task_1$	$RPE_0 \leftrightarrow Node_1$, $RPE_1 \leftrightarrow Node_1$, $RPE_0 \leftrightarrow Node_2$.	*User-defined hardware configuration* **OR** *Device-specific hardware*
$Task_2$	$RPE_1 \leftrightarrow Node_1$, $RPE_0 \leftrightarrow Node_2$.	*User-defined hardware configuration* **OR** *Device-specific hardware*
$Task_3$	$RPE_0 \leftrightarrow Node_0$.	*Device-specific hardware*

Table 3.2: Possible node mappings for tasks $Task_0$, $Task_1$, $Task_2$, and $Task_3$.

rent status of the nodes. By considering parameters as well as the right scheduling strategy, more performance gain can be achieved by utilizing reconfigurable fabric for particular applications. Moreover, the recent and future reconfigurable devices are expected to support dynamic partial reconfigurability, and different hardware implementations on the same RPE are possible. With the proposed virtualization framework, the resources can be managed in a more efficient manner when the available processing elements are both GPPs and RPEs. Those distributed computing applications which contain more parallelism can get more benefit if executed on the reconfigurable hardware. For the purpose of testing task scheduling strategies and resource management for dynamic reconfigurable processing nodes in a distributed environment, we have developed a simulation framework, termed as **D**ynamic **Re**configurable **A**utonomous **M**any-task **Sim**ulator (DReAMSim) [50]. Moreover, we extended the simulator to add partial reconfigurable functionality to the nodes in the DReAMSim [51]. The design offers to model complex reconfigurable computing nodes, processor configurations, and tasks along with GPPs. A number of different reconfiguration parameters can be exploited to model the processor configurations. The DReAMSim can be used to investigate the desired system scenario(s) for a particular scheduling strategy and a given number of tasks, computing nodes, configurations, task arrival distributions, area ranges, and task required times etc. In the following chapters, we will discuss the details of DReAMSim.

3.6 Summary

This chapter presented a generic virtualization framework for RPEs in distributed computing systems. We presented various scenarios in terms of use-cases to discuss the utilization of RPEs in distributed computing systems. Based on different virtualization levels, we provided a general model for a computing node which integrates both GPPs and RPEs in distributed computing systems. We also presented a typical application task model. Finally, we presented a case study of a scientific application which can take benefit from our virtualization framework. The proposed framework is generic in nature and provides *backward compatibility* for existing software applications in the system. Moreover, it is adaptive in dynamically adding or removing resources. Independent of the virtualized hardware, the framework can be extended different types of *services* to the user, such as monitoring and status updates etc.

Note.

The content of this chapter is based on the following paper:

M. F. Nadeem, M. Nadeem, and S. Wong. **On Virtualization of Reconfigurable Hardware in Distributed Systems**. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW 2012)*, pp. 9, Pittsburgh, PA, USA, September 2012.

4

Resource Information Services

In previous chapters, we outlined the necessary background information on the importance of novel virtualization frameworks and simulation tools to utilize **R**econfigurable **P**rocessing **E**lements (RPEs) in distributed systems. In Chapter 3, we provided a detailed description of a virtualization framework to integrate RPEs in the computing nodes of a distributed computing system. In this chapter, we describe a detailed motivational example to highlight the resource information services required to maintain the information regarding RPEs in distributed systems. Resource management is a core feature in the dynamic environment of a distributed computing system. Fundamentally, a **R**esource **I**nformation **S**ystem (RIS) provides services to maintain the updated information of all the resources in the system. This information—which can be of *dynamic* as well as *static* nature—plays a key role in the resource discovery, allocation and task scheduling. The management of this information becomes a more challenging task if the system contains dynamic and heterogeneous resources, such as Reconfigurable Processing Elements (RPEs). In this chapter, we describe the resource management scenarios in a distributed system, in the context of adding RPEs as a computing resource. We provide some basic concepts in this respect, and propose a formulation of the system model. Furthermore, we describe a simple mechanism to maintain the information regarding the resource nodes in a distributed system containing RPEs. We discuss an RIS, that contains data structures to update the information required by the other modules—such as, the scheduler, the load balancing system, the monitoring system—in resource management. These data structures contain the information corresponding to all the nodes in the system, that includes the updated statuses of the processor configurations on each node, available reconfigurable areas of the nodes, the currently running tasks, and the current state of the nodes. Finally, we provide a detailed example to explain the functionality of these data structures in an RIS.

4.1 Resource Information Services—Basic Concepts

Resource management is one of the most important elements of a distributed computing system [112] [113]. Its fundamental functionality is to identify the requirements of a job, discover and allocate resources to it, assist in its scheduling, and monitor the resources to avoid any possible failures. By performing the coordinated management and the virtualization of resources, the overall resource utilization efficiency can be improved. A **R**esource **I**nformation **S**ystem (RIS) is a major ingredient in the management of resources in a distributed system [114] [115] [116] [117] [118]. It collects and integrates all the information in the system, and provides a unified access to a user. In a complex system, such information can be *static* as well as *dynamic*. For instance, the *static* information consists of the memory sizes, the architectures, the FPGA device family, and the total reconfigurable area, etc. Similarly, the *dynamic* information includes the current state of the resources, the current task(s) running on a particular resource, the available reconfigurable area on the node, etc. In order to maintain all this information, an RIS interacts closely with various other systems such as, a *task scheduling system*, a *task submission system*, and a *task analyzer*. As depicted in Figure 4.1, when a task is submitted to the task submission system, it is analyzed for an appropriate resource selection. The process of resource selection is dependent on the precise information provided by the RIS. Similarly, a *task scheduling system* or a *scheduler* is the core part of a typical distributed computing system. It receives job/task requests and chooses appropriate nodes to execute them. A task scheduling algorithm determines which scheduling strategy to use for resource matching. In a dynamic computing environment, due to the diversity of resources and their large numbers, the monitoring of resources and providing the respective services are significant and complex challenges. The key objectives of a *Resource Information System* are as follows:

- Assisting in resource allocation and discovery.
- Providing the user applications with a unified access to the resources.
- Isolating the application tasks from the underlying hardware resources.
- Adding compatibility to the distributed system by allowing the matchmaking of the tasks to those resources that are not the best-match options, but still compatible to them.

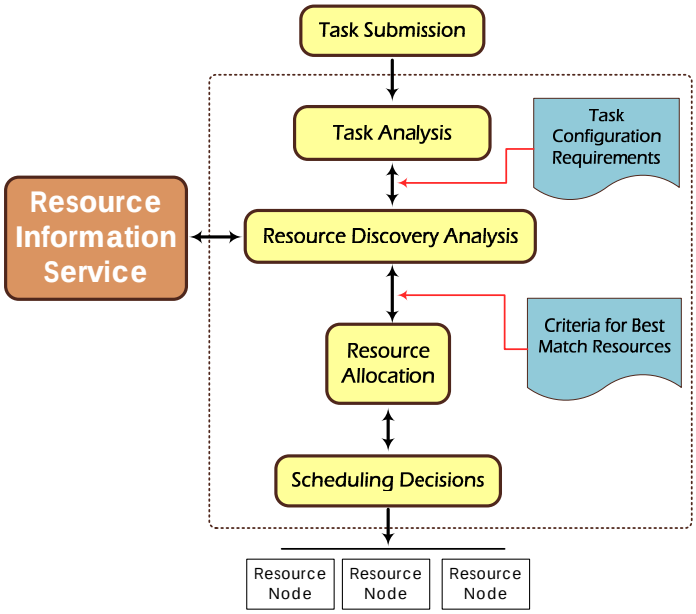


Figure 4.1: Resource Information Service in distributed computing systems.

- Minimizing the overheads related to the match-making.

The incorporation of RPEs—in order to gain more performance and flexibility—in a distributed computing system, further increases the above-mentioned design complexity of an RIS. At first, they introduce new design parameters—such as, reconfigurable area, reconfiguration overheads, and context-switching etc—that need to be addressed in updating the system. Because the resource discovery and allocation might depend on these parameters related to a reconfigurable node. Secondly, their dynamic behavior is dissimilar from that of GPPs, as they have a different set of requirements in order to execute an application task. Finally, their performance depends on the amenability of the application to be accelerated on a particular resource. It highly depends on the intrinsic parallelism of the application. In this scenario, new system models are required to formulate the components and the functionality of an RIS.

Figure 4.2 depicts a conceptual overview of a large-scale distributed system containing reconfigurable processing nodes, as well as GPPs. It consists of a *Resource Management System (RMS)* which handles the monitoring, load distribution, application task scheduling among different nodes in the system.

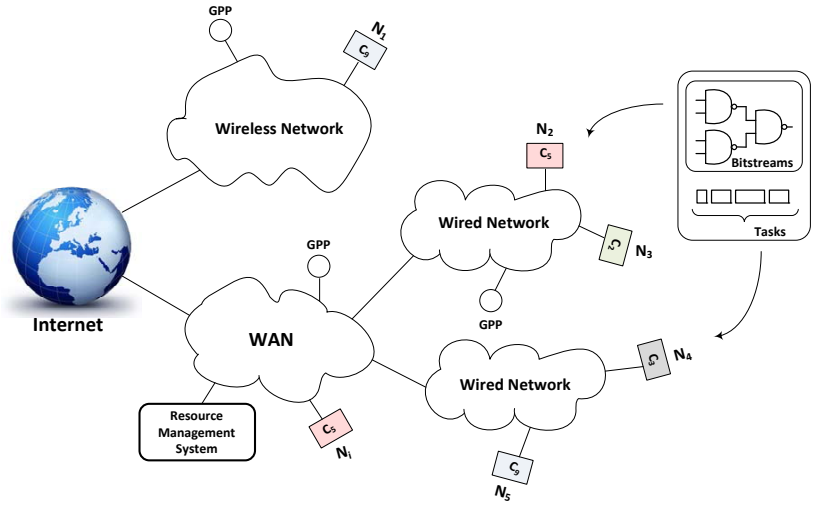


Figure 4.2: A conceptual overview of a distributed system with reconfigurable nodes.

All these functionalities require important and updated information provided by an RIS. It can be noted that each reconfigurable node contains a particular configuration (denoted as C_i) of a processor of a certain type (P_{type}). It can execute an application task, which requires a preferred processor configuration (denoted as C_{pref}). An existing C_i on a node can be changed by sending a bitstream of a different configuration. The *RMS* distributes the tasks onto suitable nodes specified by some task scheduling algorithm. If a task prefers a certain processor configuration (C_{pref}), then the *RMS* reconfigures a suitable node by sending the corresponding bitstream and sends the task for processing. In Section 4.3, we discuss a formal system model of such a distributed computing system. Moreover, we elaborate on how the resource nodes are maintained and their respective information is updated in an RIS.

The remainder of this chapter is divided into the following sections. Section 4.2 presents some previous work on the resource information management in the distributed systems. Section 4.3 presents the formulation of the system model. It discusses formal models of the reconfigurable node, the processor configuration, and the application task. In Section 4.4, we provide a detailed example to explain the functionality of an RIS for a distributed computing system. Finally, Section 3.6 discusses the summary of this chapter.

4.2 RIS Mechanisms in Literature

In the literature, we can find several RIS mechanisms for the computing systems. A detailed taxonomy and survey of resource management systems for distributed computing is given in [112]. Moreover, another survey article provides a summary of various resource information services [114]. In the following paragraphs, we give an overview of these mechanisms.

K. Czajkowski *et al.* [118] propose a grid information system for distributed resource sharing. Their main objective is to distribute the information providers, which can access information from different entities in the system. It uses various protocols, and the architecture is based on the famous middleware, Globus Toolkit [21], which provides a software infrastructure for building computational grids that allow distributed computing cycles, scientific instruments, storage devices, and other tools to be shared securely across geographic and institutional limits. Similarly, Z. Qian *et al.* [116] also adopt Globus Toolkit models to propose a fault-tolerant resource discovery and allocation.

A. AuYoung *et al.* [119] address the resource allocation issue for federated distributed systems with heterogeneous resources, in the context of overall computing economy. The users can choose the resources of interest using a the proposed resource discovery mechanism, and express resource preferences over time and space. Resource allocation is controlled by centralized auctioneer.

D. Puppin *et al.* [115] adopt a peer-to-peer approach to design a grid information service. In this work, the resource nodes are grouped into clusters where each cluster includes a *SuperPeer* node, which collects all the information relating to the cluster.

In the domain of large-scale embedded system, Z. Pohl *et al.* [120] propose the resource manager for the heterogeneous arrays of hardware accelerators. The focus is on efficient resource utilization, power efficiency, and scalability. The resource manager is modeled for a single set of hardware accelerators.

M. Ahmadi *et al.* [96] demonstrate a simulation model of a distributed system containing RPEs. Based on the GridSim [45] simulator, it provides a model of a node that comprises of a General Purpose Processor (GPP) augmented with one or more RPEs. The GPP serves as a resource manager in this design, and it takes care of the reconfiguration of RPEs, the allocation of tasks, and adopts a collaboration with its *neighboring* nodes. The work does not give any formal model of resource management as such, because the nodes are

modeled to collaborate with each other, based on the *neighborhood* concept.

In [6], traditional distributed computing concepts are integrated with reconfigurable hardware to break a cryptographic algorithm by distributing generic VHDL code implementations to different reconfigurable nodes on a TCP/IP network. To accomplish this, a platform-independent, scalable and efficient hardware brute force key searcher is demonstrated on a network with computing nodes containing various types of FPGA boards. The resource management is performed through an online system, but it is very restricted, and only specific to the system.

4.3 Formulation of System Model

In this section, we provide a formal system model by defining node, configuration, and task models. A typical reconfigurable node can be defined by the following tuple:

$$\begin{gathered} Node_i \left(TotalArea, AvailableArea, \mathbf{C}, family, caps, state \right) \\ where \;\; \mathtt{C} = \{C_1, C_2, \cdots, C_m\} \end{gathered} \tag{4.1}$$

Where $\mathtt{C}$ represents a set of current processor configuration on the node and i represents the *node number*. $TotalArea$ is the total reconfigurable area of the node i, whereas, $AvailableArea$ is the remaining reconfigurable area on the node, after it is configured with m configurations. A device $family$ defines the group of compatible nodes which share similar types of resources and performance. Furthermore, $caps$ represent a list of different *capabilities* available on a node. For example, a node's $caps$ may include hardware resources, such as embedded memory, DSP slices, configuration bandwidth, etc. Finally, $state$ represents the status of the node i: busy or idle. Subsequently, the general configuration of a processor to be configured on a node, is represented as:

$$\begin{gathered} \mathrm{C}_i \left(ReqArea, P_{type}, \mathtt{param}, BSize, ConfigTime \right) \\ where \;\; \mathtt{param} = \{parameter_1, \cdots, parameter_k\} \end{gathered} \tag{4.2}$$

Where i represents the *configuration number* and $ReqArea$ is the total reconfigurable area required by the configuration i. P_{type} represents the processor configuration required by tasks and $\mathtt{param}$ is a set of $parameters$ representing a list of attributes of a particular P_{type} processor which provide its

Terminology	Description
Node	Defined in Tuple 3.1.
Configuration	Defined in Tuple 4.2.
Task	Defined in Tuple 4.3.
AvailableArea	Defined in Equation 4.4.
Full reconfiguration	Only one configuration per node is allowed, at a given time.
Partial reconfiguration	Multiple configurations per node are allowed, at a given time.
Idle List	Linked list of idle nodes containing the same configuration.
Busy List	Linked list of busy nodes containing the same configuration.
Sus List	List of the tasks currently put in suspension by the scheduler.
Idle node	Contains at least one configuration, but currently no task is running on it.
Busy node	Contains at least one configuration, and at least one task is currently running on it.
Blank node	Currently contains no configuration.
Full node	Contains maximum configurations, and all of them are running tasks. No further configurations are possible due to area limitations.
Preferred configuration (C_{pref})	A preferred processor configuration required by a certain task.
Closest configuration ($C_{closest}$)	Closest possible configuration to the C_{pref}, based on some criteria.
Best-match node	Best node match for a given task, based on some criteria.
Best partially blank node	Best node match (based on some criteria) among those nodes that already contain at least one configuration.

Table 4.1: Definitions of some important terminologies.

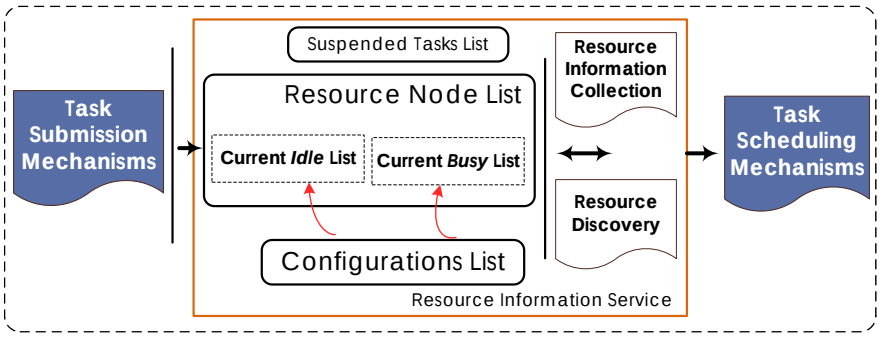


Figure 4.3: The RIS data structures.

architectural details. Some examples of P_{type} are multipliers, systolic arrays, soft-core processors, and custom-made signal processors. One such example is a *parameterizable* soft-core ρ-VEX VLIW processor presented in [121]. It has been implemented on FPGA and can be adopted to several architectural `parameters`. These `parameters` include the number and types of functional units (multipliers and ALUs), cluster cores, the number of issues, or the number of memory slots. Finally, `BSize` represents the file size of bitstream for a configuration C_i. Similarly, an application task is defined by the following tuple:

$$Task_i \left(t_{required}, C_{pref}, data\right) \tag{4.3}$$

Where i represents the *task number*, $t_{required}$ is the execution time required by the task i if it is processed on its preferred processor configuration (C_{pref}), and `data` is the input data of the task. C_{pref} is the preferred processor configuration required by task i. This configuration is a specific processor implemented on a reconfigurable node.

Based on the definitions of above tuples, the `AvailableArea` of a node can be calculated as follows:

$$AvailableArea = \begin{cases} TotalArea & \text{if } m = 0 \\ TotalArea - \sum_{i=1}^{m} ReqArea_i & \text{if } m > 0 \end{cases} \tag{4.4}$$

Where m represents the cardinality of the set C in Tuple 4.1, and it provides the total number of current configurations of the node. Table 4.1 defines all the important terminologies used in the next pages.

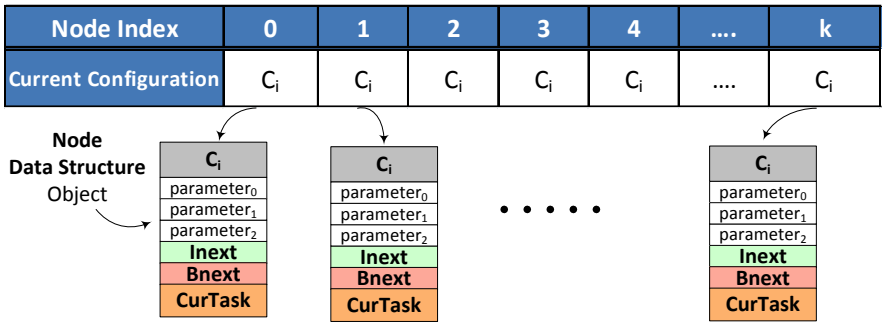


Figure 4.4: The *nodes list* in the *Resource Information System.*

4.3.1 Data Structures in RIS

The core of an RIS are the data structures which update the current information of all the nodes. Figure 4.3 depicts these data structures denoted as the *node list* and *configuration list*. The *node list* updates the information related to all the reconfigurable nodes in the system, as depicted in Figure 4.4. Each object of this dynamic data structure contains the updated information about the current configuration (C_i), and currently running task (*CurTask*), while the static information consists of various other parameters, such as, FPGA device family, area, and *caps* etc. Table 3.1 contains a list of similar parameters for different processing elements. Notably, each object in the *nodes list* contains pointers, denoted by *Inext* and *Bnext*. These pointers link a node to one or more nodes that contain the same configuration as the current node. In this way, all the nodes with a particular configuration form a list of nodes, which is termed as the *Busy List* or the *Idle List*. The *nodes list* data structure also maintains the details of other dynamic parameters, such as the available reconfigurable area of the node. These data structures are explained in more detailed way in next chapters.

In an RIS, these data structures work closely with the task scheduler to provide a precise information for the resource discovery and allocation. In the next section, we give a detailed example of various node lists being maintained by these data structures, by discussing several scenarios.

4.4 RIS Motivational Example

Before discussing the details of the example, we provide some terminologies, assumptions, and notations used in the explanation. Most of the terminologies are summarized in Table 4.1. A *full reconfiguration* scenario means that a node can only be reconfigured for a single configuration at one time. It can also be reconfigured for a different configuration, only if the node becomes idle after finishing the processing of its current task. In this case, an *Idle* node is the one that contains a particular configuration, but currently holding no task for execution, while a *Busy* node is currently processing a task. On the other hand, a *partial reconfiguration* scenario allows a node to reconfigure as many configurations as possible, if there is sufficient area available on the node. In this case, an *Idle* node may contain more than one configuration, but it holds no tasks for execution, while a *Busy* node contains at least one currently running task.

We consider a distributed computing system containing RPEs, that is similar to the one depicted in Figure 4.2. These nodes are maintained by an RIS, and the core of this system are the data structures that update the dynamic statuses of all the nodes. These nodes are well-connected by an efficient network system. It is considered that a task scheduler, running a specific scheduling algorithm, distributes tasks among the nodes. For the time being, we ignore the details of the scheduling algorithm. Furthermore, we presume that, before assigning a task to it, the scheduler takes care of the area requirements on each node. The RIS maintains the statuses of the current configuration on a particular node, available areas of the nodes, and the *Current busy* and *Idle lists* specific to each node, in the defined data structures. It is presumed that the tasks (denoted by $\mathtt{T_i}$) are injected into the distributed system for execution on the nodes by the task scheduler. When a task is assigned to a node for execution, it becomes a *Busy node*. When a task finishes its execution, the corresponding node becomes an *Idle node*. Due to this dynamic behavior of the nodes, it is important to maintain the current status of each node in order to assign new tasks. These statuses can be maintained by linking the nodes of the same configuration together. The list manipulations are detailed throughout the example. Referring to Figure 4.5, each node n_i contains a pair represented as $(C_i\,,\mathtt{T_i})$, where C_i is the current configuration on the node, and $\mathtt{T_i}$ represents the current task running on the node, utilizing C_i. If there is no task currently being executed on the node, it is represented as $(C_i\,,\Phi)$.

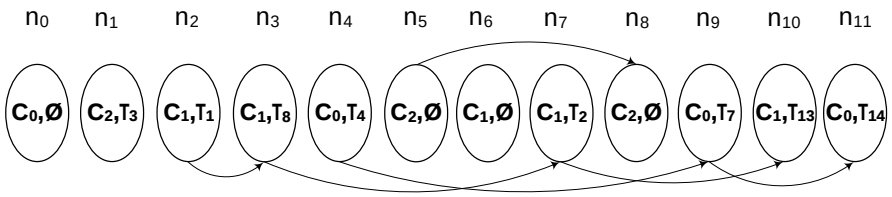


Figure 4.5: Current *Idle* and *Busy* lists of the system DS_1 at time $t = t_1$ (Initial condition).

Config.	Current *Idle* list	Current *Busy* list
C_0	n_0	$n_4(T_4) \rightarrow n_9(T_7) \rightarrow n_{11}(T_{14})$
C_1	n_6	$n_2(T_1) \rightarrow n_3(T_8) \rightarrow n_7(T_2) \rightarrow n_{10}(T_{13})$
C_2	$n_5 \rightarrow n_8$	$n_1(T_3)$

Table 4.2: Current *Idle* and *Busy* lists of all configurations of the system DS_1 at time $t = t_1$ (One configuration per node allowed).

4.4.1 One Configuration per Node Allowed

Let us consider a distributed computing system, denoted by DS_1. In this system, each reconfigurable node is allowed to be configured with only one configuration at a given time. We assume that, initially, we have only 3 configurations (denoted by C_0, C_1, and C_2) in the DS_1 that contains 12 reconfigurable nodes, each with a fixed area. We represent these nodes by $n_0, n_1, n_2, \ldots\ldots, n_{11}$. We discuss various scenarios in the following.

Initial condition of DS_1 (at time $t = t_1$)

Figure 4.5 depicts an initial condition of the system DS_1 at a given time $t = t_1$. It can be noticed that all the idle nodes with the same configuration are linked together by maintaining the corresponding *current idle list*. These links are represented by arrows on top of the nodes. For instance, nodes n_5 and n_8 are currently not executing any tasks, but they are configured with C_2. Therefore, the *current idle list* with the configuration C_2 can be represented as $n_5 \rightarrow n_8$. Similarly, all the busy nodes with a certain configuration are linked together by maintaining the corresponding *current busy list*. These links are represented by arrows on bottom of the nodes. For instance, n_2, n_3, n_7 and n_{10} are currently configured with C_1. Each of these nodes is busy in executing a certain task. Therefore, the *current busy list* with the configuration C_1 is

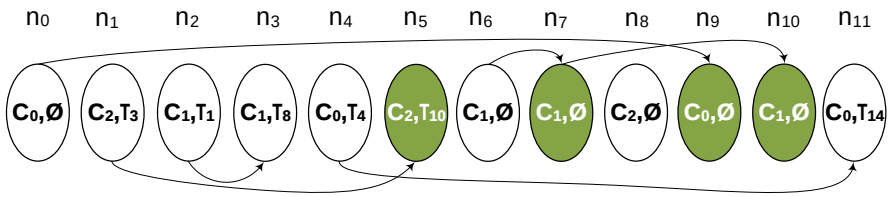


Figure 4.6: Current *Idle* and *Busy* lists of the system DS_1 at time $t = t_2$ (some nodes (n_7, n_9, and n_{10}) are idle now).

Config.	Current *Idle* list	Current *Busy* list
C_0	$n_0 \rightarrow n_9$	$n_4(T_4) \rightarrow n_{11}(T_{14})$
C_1	$n_6 \rightarrow n_7 \rightarrow n_{10}$	$n_2(T_1) \rightarrow n_3(T_8)$
C_2	n_8	$n_1(T_3) \rightarrow n_5(T_{10})$

Table 4.3: Current *Idle* and *Busy* lists of all configurations of the system DS_1 at time $t = t_2$ (One configuration per node allowed).

represented as $n_2(T_1) \rightarrow n_3(T_8) \rightarrow n_7(T_2) \rightarrow n_{10}(T_{13})$. The status of all the current idle and busy lists at time $t = t_1$, for each configuration are given in Table 4.2.

Idle nodes (at time $t = t_2$)

Since DS_1 is a dynamic system, it changes its state from time t_1 to t_2. Figure 4.6 depicts the state of the system DS_1 at time $t = t_2$. At this time, the nodes n_7, n_9, and n_{10} have become idle after executing their corresponding tasks. Moreover, the node n_5 becomes busy in executing task T_{10} being assigned by the scheduler. Table 4.3 shows the corresponding busy and idle lists after these changes in the statuses of the nodes in the system DS_1 at time $t = t_2$. It can be noticed that the *current idle list* for configuration C_1 has changed from n_5 to $n_6 \rightarrow n_7 \rightarrow n_{10}$. Similarly, the corresponding *current busy list* has changed to $n_2(T_1) \rightarrow n_3(T_8)$, because the nodes n_7 and n_{10} have become idle.

Reconfiguration of nodes (at time $t = t_3$)

Figure 4.7 depicts the state of the system DS_1 at time $t = t_3$. At this time, the scheduler decides to assign 3 new tasks T_9, T_{11}, and T_{15} to the system DS_1.

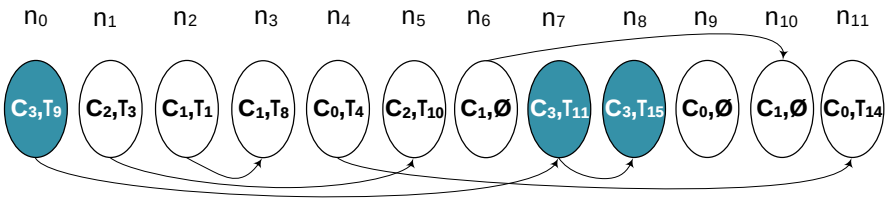


Figure 4.7: Current *Idle* and *Busy* lists of the system DS_1 at time $t = t_3$ (some nodes (n_0, n_7, and n_8) are reconfigured with a new configuration (C_3).

Config.	Current *Idle* list	Current *Busy* list
C_0	n_9	$n_4(T_4) \rightarrow n_{11}(T_{14})$
C_1	$n_6 \rightarrow n_{10}$	$n_2(T_1) \rightarrow n_3(T_8)$
C_2	**NULL**	$n_1(T_3) \rightarrow n_5(T_{10})$
C_3	**NULL**	$n_0(T_9) \rightarrow n_7(T_{11}) \rightarrow n_8(T_{15})$

Table 4.4: Current *Idle* and *Busy* lists of all configurations of the system DS_1 at time $t = t_3$(One configuration per node allowed).

Each of these nodes requires a new configuration C_3. Thus, the scheduler chooses 3 idle nodes (n_0, n_7 and n_8) and reconfigures these nodes with the configuration C_3. After the assignment of the new tasks, the current busy and idle lists in the system DS_1 are updated accordingly, as shown in Table 4.4. It can be noticed that new idle and busy lists for the configuration C_3 are created. Since, no node is currently idle with configurations C_2 and C_3, so the corresponding idle lists are represented by **NULL**. The *current busy list* for the new configuration C_3 is now created as $n_0(T_9) \rightarrow n_7(T_{11}) \rightarrow n_8(T_{15})$.

4.4.2 Multiple Configurations per Node Allowed

For this scenario, we consider a distributed computing system—denoted by DS_2—in which the nodes are allowed to reconfigure multiple configurations at a given time, depending upon the availability of sufficient area. We assume that, initially, we have only 4 configurations (denoted by C_0, C_1, C_2, and C_3) in DS_2 with 12 reconfigurable nodes, represented by $n_0, n_1, n_2, \ldots\ldots, n_{11}$. These nodes allow *partial reconfiguration*, which implies that a certain node region can be configured on runtime, while it is already configured with some configurations on its other regions. In this example, we discuss the status of the system DS_2 at 4 different time instances t_0, t_1, t_2, and t_3. Referring

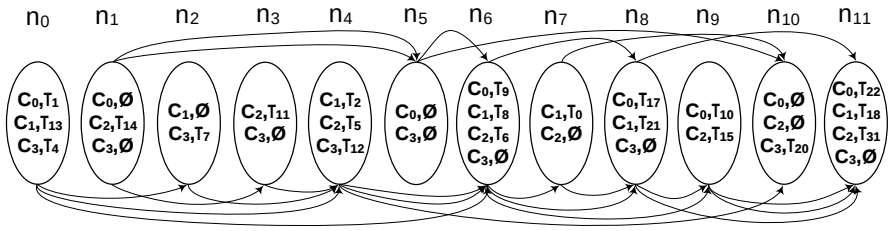


Figure 4.8: Current *Idle* and *Busy* lists of the system $\mathtt{DS}_2$ at time $t = t_1$ (Initial condition).

Config.	Current *Idle* list	Current *Busy* list
C_0	$n_1 \rightarrow n_5 \rightarrow n_{10}$	$n_0(\mathtt{T}_1) \rightarrow n_6(\mathtt{T}_9) \rightarrow n_8(\mathtt{T}_{17}) \rightarrow n_9(\mathtt{T}_{10}) \rightarrow n_{11}(\mathtt{T}_{22})$
C_1	n_2	$n_0(\mathtt{T}_{13}) \rightarrow n_4(\mathtt{T}_2) \rightarrow n_6(\mathtt{T}_8) \rightarrow n_8(\mathtt{T}_{21}) \rightarrow n_{11}(\mathtt{T}_{18})$
C_2	$n_7 \rightarrow n_{10}$	$n_1(\mathtt{T}_{14}) \rightarrow n_3(\mathtt{T}_{11}) \rightarrow n_4(\mathtt{T}_5) \rightarrow n_6(\mathtt{T}_6) \rightarrow n_9(\mathtt{T}_{15}) \rightarrow n_{11}(\mathtt{T}_{31})$
C_3	$n_1 \rightarrow n_5 \rightarrow n_6 \rightarrow n_8 \rightarrow n_{11}$	$n_0(\mathtt{T}_4) \rightarrow n_2(\mathtt{T}_7) \rightarrow n_4(\mathtt{T}_{12}) \rightarrow n_{10}(\mathtt{T}_{20})$

Table 4.5: Current *Idle* and *Busy* lists of all configurations of the system $\mathtt{DS}_2$ at time $t = t_1$ (Multiple configurations per node allowed).

to Figure 4.8, each node n_i contains multiple pairs represented as $(C_i\,,\,\mathtt{T_i})$, where C_i is the current configuration on the node, and $\mathtt{T_i}$ represents the current task running on the node, utilizing C_i.

Initial condition of $\mathtt{DS}_2$ (at time $t = t_1$)

Figure 4.8 depicts the linked lists of nodes with multiple configurations in system $\mathtt{DS}_2$ at a certain time $t = t_1$. It can be noticed that all the idle nodes with the same configurations are linked together by maintaining the corresponding *current idle list*. These links are represented by arrows on top of the nodes. For instance, nodes n_7 and n_{10} are currently not executing any tasks, but they are configured with the configuration C_2. Therefore, the *current idle list* of nodes with configuration C_2 is $n_7 \rightarrow n_{10}$. Similarly, all the busy nodes with a certain configuration are linked together, represented by arrows on bottom of those nodes. Referring to Figure 4.8, it can be noticed that each node contains multiple $(C_i\,,\,\mathtt{T_i})$ pairs. It means that, in this case, a node can

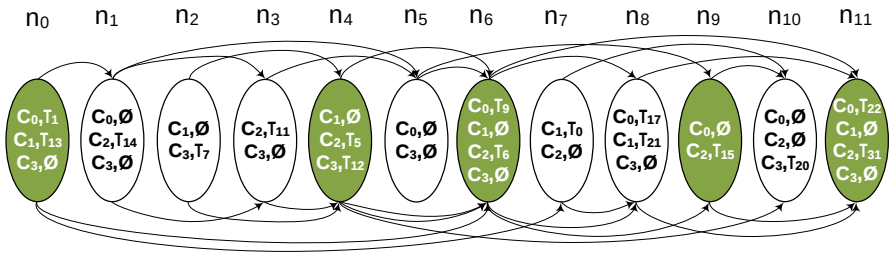


Figure 4.9: Current *Idle* and *Busy* lists of the system DS_2 at time $t = t_2$ (regions in some nodes (n_0, n_4,n_6, n_9, and n_{11}) are idle now).

Config.	Current *Idle* list	Current *Busy* list
C_0	$n_1 \rightarrow n_5 \rightarrow n_9 \rightarrow n_{10}$	$n_0(T_1) \rightarrow n_6(T_9) \rightarrow n_8(T_{17}) \rightarrow n_{11}(T_{22})$
C_1	$n_2 \rightarrow n_4 \rightarrow n_6 \rightarrow n_{11}$	$n_0(T_{13}) \rightarrow n_7(T_0) \rightarrow n_8(T_{21})$
C_2	$n_7 \rightarrow n_{10}$	$n_1(T_{14}) \rightarrow n_3(T_{11}) \rightarrow n_4(T_5) \rightarrow n_6(T_6) \rightarrow n_9(T_{15}) \rightarrow n_{11}(T_{31})$
C_3	$n_0 \rightarrow n_1 \rightarrow n_3 \rightarrow n_5 \rightarrow n_6 \rightarrow n_8 \rightarrow n_{11}$	$n_2(T_7) \rightarrow n_4(T_{12}) \rightarrow n_{10}(T_{20})$

Table 4.6: Current *Idle* and *Busy* lists of all configurations of the system DS_2 at time $t = t_2$ (Multiple configurations per node allowed).

become part of multiple idle and busy lists. In this way, if 2 different node regions of a certain node n_i are configured with 2 different configurations, and both are active in executing tasks, then the node n_i is part of 2 busy lists of corresponding configurations. For instance, in Figure 4.8, node n_0 contains 3 different configurations (C_0, C_1, and C_3) and it is part of 3 different busy lists as shown in Table 4.5. Nodes n_1, n_5, and n_{10} are configured with C_0, but are currently not running any tasks on C_0, so these nodes constitute the *current idle list* of C_0, as $n_1 \rightarrow n_5 \rightarrow n_{10}$. The corresponding *current busy list* of C_0 is $n_0(T_1) \rightarrow n_6(T_9) \rightarrow n_8(T_{17}) \rightarrow n_9(T_{10}) \rightarrow n_{11}(T_{22})$. Table 4.5 provides all the possible busy and idle lists of each configuration in the system DS_2 at time $t = t_1$.

Idle node regions (at time $t = t_2$)

At time $t = t_2$, regions in some nodes (n_0, n_4, n_6, n_9, and n_{11}) have become idle after completing the execution of their respective tasks. Figure 4.9 depicts the current lists of the system DS_2 at time $t = t_2$. These node regions

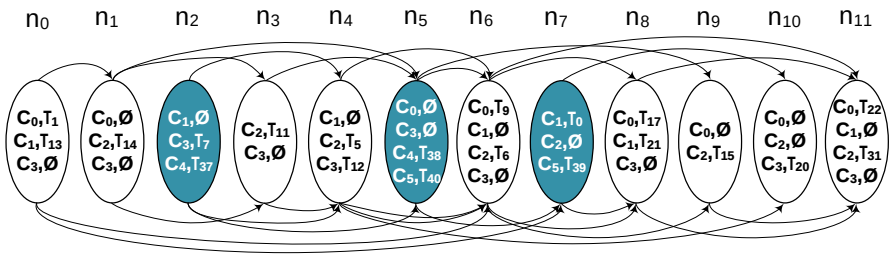


Figure 4.10: Current *Idle* and *Busy* lists of the system DS_2 at time $t = t_3$ (regions in some nodes (n_2, n_5, and n_7) are configured with new configurations C_4 and C_5).

Config.	Current *Idle* list	Current *Busy* list
C_0	$n_1 \rightarrow n_5 \rightarrow n_9$	$n_0(T_1) \rightarrow n_6(T_9) \rightarrow n_8(T_{17}) \rightarrow n_{11}(T_{22})$
C_1	$n_2 \rightarrow n_4 \rightarrow n_6 \rightarrow n_{11}$	$n_0(T_{13}) \rightarrow n_7(T_0) \rightarrow n_8(T_{21})$
C_2	$n_7 \rightarrow n_{10}$	$n_1(T_{14}) \rightarrow n_3(T_{11}) \rightarrow n_4(T_5) \rightarrow n_6(T_6) \rightarrow n_9(T_{15}) \rightarrow n_{11}(T_{31})$
C_3	$n_0 \rightarrow n_1 \rightarrow n_3 \rightarrow n_5 \rightarrow n_6 \rightarrow n_8 \rightarrow n_{11}$	$n_2(T_7) \rightarrow n_4(T_{12}) \rightarrow n_{10}(T_{20})$
C_4	NULL	$n_2(T_{37}) \rightarrow n_5(T_{38})$
C_5	NULL	$n_5(T_{40}) \rightarrow n_7(T_{39})$

Table 4.7: Current *Idle* and *Busy* lists of all configurations of the system DS_2 at time $t = t_3$ (Multiple configurations per node allowed).

(nodes are highlighted in Figure 4.9) can receive further tasks assigned by the scheduler. With these changes, the *current busy list* and the *current idle list* of each configuration are updated accordingly, and they are enlisted in Table 4.6. For instance, n_4, n_6 and n_{11} have become idle with configuration C_1, and the corresponding *current idle list* is updated from n_2 to $n_2 \rightarrow n_4 \rightarrow n_6 \rightarrow n_{11}$. Similarly, the *current busy list* of C_1 is now $n_0(T_{13}) \rightarrow n_7(T_0) \rightarrow n_8(T_{21})$.

Configuration of node regions (at time $t = t_3$)

Figure 4.10 depicts the current lists of the system DS_2 at time $t = t_3$. At this time, the scheduler wants to assign new tasks T_{37}, T_{38}, T_{39}, and T_{40}, requiring configurations C_4 and C_5. However, no node regions in DS_2 are currently configured with these new configurations. Therefore, the scheduler finds some nodes (n_2, n_5, and n_7) that are already configured with some configurations,

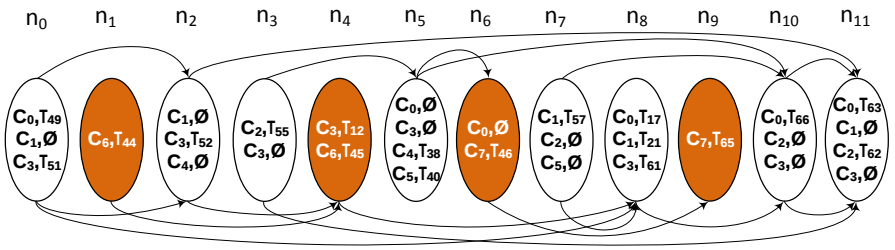


Figure 4.11: Current *Idle* and *Busy* lists of the system DS_2 at time $t = t_4$ (some nodes (n_1, n_4, n_6 and n_7) are *partially* or *fully* reconfigured with new configurations C_6 and C_7).

Config.	Current *Idle* list	Current *Busy* list
C_0	$n_5 \rightarrow n_6$	$n_0(T_{44}) \rightarrow n_8(T_{17}) \rightarrow n_{10}(T_{66}) \rightarrow n_{11}(T_{63})$
C_1	$n_0 \rightarrow n_2 \rightarrow n_{11}$	$n_7(T_{57})$
C_2	$n_7 \rightarrow n_{10}$	$n_3(T_{55}) \rightarrow n_{11}(T_{62})$
C_3	$n_3 \rightarrow n_5 \rightarrow n_{10} \rightarrow n_{11}$	$n_0(T_{51}) \rightarrow n_2(T_{52}) \rightarrow n_4(T_{12}) \rightarrow n_8(T_{61})$
C_4	n_2	$n_5(T_{38})$
C_5	n_7	$n_5(T_{40})$
C_6	**NULL**	$n_1(T_{44}) \rightarrow n_4(T_{45})$
C_7	**NULL**	$n_6(T_{46}) \rightarrow n_9(T_{65})$

Table 4.8: Current *Idle* and *Busy* lists of all configurations of the system DS_2 at time $t = t_4$ (Multiple configurations per node allowed).

but contain sufficient area to accommodate new configurations. These nodes are partially configured with C_4 and C_5, and the new tasks are assigned to them. At time $t = t_3$, there is no idle node region in DS_2 with configurations C_4 and C_5, so their *current idle lists* are **NULL**. However, the corresponding *current busy lists* are $n_2(T_{37}) \rightarrow n_5(T_{38})$ and $n_5(T_{40}) \rightarrow n_7(T_{39})$ for C_4 and C_5, respectively. The current idle and busy lists for each configuration are given in Table 4.7.

Reconfiguration of nodes (at time $t = t_4$)

At time $t = t_4$, the scheduler wants to assign tasks which require new configurations C_6 and C_7. However, there are no nodes available with sufficient area to accommodate these new configuration. Therefore, the scheduler decides to *reconfigure* some nodes by removing some (or all) of their existing

idle configurations. Figure 4.11 depicts that the nodes n_1, n_4, n_6, and n_9 are reconfigured with new configurations C_6 and C_7. It can be noticed that all the previous configurations (as depicted in Figure 4.10) are removed from the nodes n_1 and n_9; whereas, the nodes n_4 and n_6 still contain some previous configurations (C_3 and C_0). For instance, n_1 finished executing its task T_{14}, and all its region became idle. Therefore, all the existing configurations are removed to reconfigure it with configuration C_6. On the other hand, node n_4 finished executing T_5, and it has two idle node regions with configurations C_1 and C_2. These configurations are removed and C_6 is reconfigured, keeping an already active configuration C_3 which is busy in executing its task T_{12}. In this case, the current lists are completely re-adjusted as depicted in Figure 4.11. For example, the *current idle list* for configuration C_0 is now $n_5 \rightarrow n_6$; whereas, the *current busy list* for this case is updated as $n_0(T_{44}) \rightarrow n_8(T_{17}) \rightarrow n_{10}(T_{66}) \rightarrow n_{11}(T_{63})$. Similarly, the current idle lists for the new configurations C_6 and C_7 are **NULL**, and the corresponding busy lists are $n_1(T_{44}) \rightarrow n_4(T_{45})$ and $n_6(T_{46}) \rightarrow n_9(T_{65})$, respectively. Table 4.8 provides all the current busy and idle lists of each configuration at time $t = t_4$.

4.4.3 Conclusions of the Example

The above example gives an insight into the dynamic behavior of the nodes, and the possible complexity of the resource management, by detailing several scenarios in a distributed computing system. It encompasses various situations confronted by an RIS. The linked lists facilitate to reduce the search effort required to retrieve the status information of a particular node, as compared to a scenario when all this information is updated through linear arrays. It can be particularly time-consuming to retrieve status information of a node if the number of nodes is very large. These linked lists provide a simple mechanism to maintain information corresponding to dynamically changing nodes. We ignored the details of a scheduler, that in reality, performs the task allocations. Further details on the implementation of the *scheduler* and the RIS are given in the next chapters. Consequently, they provide a basis to implement the important resource management modules in the simulation framework—called the DReAMSim—discussed in the next pages.

4.5 Summary

In this chapter, we described the resource management scenarios in a distributed computing system, in the context of adding RPEs as a computing resource. We discussed some basic concepts in this respect, and proposed a formulation of a system model. We also described a simple mechanism to maintain the information regarding the resource nodes. We discussed a *Resource Information System*, that contains data structures to update the information required by the other modules—such as, the scheduler, load balancing system, etc—in the resource management. These data structures contain the information corresponding to all the nodes in the system. We provided a detailed example to explain the functionality of the proposed data structures in an RIS.

Note.

The content of this chapter is based on the following paper:

M. F. Nadeem, S. A. Ostadzadeh, M. Ahmadi, M. Nadeem, and S. Wong. **A Novel Dynamic Task Scheduling Algorithm for Grid Networks with Reconfigurable Processors**. In *Proceedings of the 5th HiPEAC Workshop on Reconfigurable Computing (WRC 2011).* In *conjunction with the 6th International Conference on High Performance and Embedded Architectures and Compiles (HiPEAC 2011)*, Heraklion, Crete, GREECE, January 2011.

5

The DReAMSim Simulation Framework

This chapter describes the details of our proposed simulation framework—termed as DReAMSim—designed to analyze the performance of application task scheduling in distributed systems, incorporating reconfigurable processors. The framework can be utilized to simulate reconfigurable computing nodes, application tasks, and processor configurations in a distributed computing environment. Various reconfiguration parameters such as, task arrival distributions, completion times, and reconfigurable area ranges can be exploited to develop a simulation environment. Consequently, it provides an opportunity to test many task scheduling strategies, for a selective set of these parameters and simulation environments. The framework integrates the data structures which have been detailed in the previous chapter. In the subsequent sections, we explain the design and implementation specifications of the framework.

5.1 Introduction

The framework is generic in nature, and implemented in a modular way to allow further extensions in the future. We termed the proposed simulation framework as the **D**ynamic **Re**configurable **A**utonomous **M**any-task **Sim**ulator (DReAMSim). As described in the previous chapters, DReAMSim is developed for the tasks scheduling and the resource management of dynamic reconfigurable computing nodes in a distributed computing environment. The design offers to simulate complex reconfigurable computing nodes, application tasks, and processor configurations, along with the General Purpose Processors (GPPs). A particular processor configuration can be modeled by exploiting different parameters, such as the required reconfigurable area, processor type, and configuration time. Similarly, a reconfig-

urable node can be modeled by defining its total reconfigurable area, device family, and a list of hardware capabilities. These capabilities may include hardware resources, such as softcore processors, DSP slices, embedded memory, and configuration bandwidth etc. Moreover, an application task can be modeled by choosing the required completion time, data, and the preferred processors. The framework can be mainly used to investigate the impact of a particular scheduling algorithm in a distributed system, that contains reconfigurable processors. In this way, a desired system scenario can be achieved for a scheduling strategy for a given set of parameters, such as nodes, configurations, tasks, node reconfigurable area ranges, task arrival distributions, task completion times, and configuration times etc. Furthermore, a user can develop a simulation environment, where the nodes can incorporate *partial* reconfigurable functionality. In this manner, a node *region* can be reconfigured dynamically, while the other node *regions* are already configured with some configurations and running tasks. In this scenario, a node can execute multiple tasks simultaneously. In Chapter 4, we detailed the design of data structures utilized in the *resource information manager* in DReAMSim, by providing a motivational example. In this chapter, we will discuss the implementation details of these data structures. Moreover, we describe the implementation of the framework using a UML model of DReAMSim.

The remainder of the chapter is divided into the following sections. Section 5.2 discusses the top-level view of the framework, and it details the various subsystems of the framework. Section 5.3 enlists and explains some important terminologies. Section 5.4 gives the details of design and implementation of the framework. It includes the discussion on the implementation of the data structures for resource information management. Moreover, it encompasses a UML model that presents various classes, and methods used in the development of the framework. Section 5.5 describes many performance metrics generated by the framework. Section 5.6 describes a simple scheduling algorithm that takes into account the reconfigurable processors for the assignment of tasks. Section 5.7 discusses the simulation conditions and results. Finally, Section 5.8 gives a brief summary of the chapter.

5.2 Top-Level Organization of the Framework

The fundamental design of the framework is based on the system model formally described in Section 4.3, in Chapter 4. Tuples 4.1, 4.2, and 4.3 formulate a typical node, configuration, and application task in the system, respectively.

Figure 5.1 depicts the top-level organization of the DReAMSim simulation framework. There are 4 different subsystems in the proposed framework; *input*, *information*, *core*, and *output* subsystems. Each subsystem is composed of various modules, which are discussed as follows.

The *input* subsystem: it is the input interface to allow user inputs to the framework. It allows a user to set the *application specifications* as well as the *user-defined resource specifications*. It generates synthetic tasks which may require a particular processor configuration (C_{pref}) and the required estimated time for the execution of tasks, as defined in Tuple 4.3. It can also support real workloads and user constraints. A user can specify the task arrival rate and arrival distribution functions. The *User-defined resource specification* module is responsible for generating nodes and different processor *configurations*. It can produce nodes with various reconfigurable area sizes. Reconfiguration method (*full* or *partial* reconfiguration) can also be set. For simulation purposes, a user can specify the node upper and lower area limits. For a more realistic study, these limits can be specified according to the real area sizes of the reconfigurable devices available in the market. Similarly, a variety of processor *configurations* can be generated. Again for a realistic experiment, different processor *configurations* of P_{type} (as defined in Tuple 4.2) and their corresponding parameters can be specified allowing various processor options for application tasks.

The *information* subsystem: this subsystem provides resource information during a particular simulation. The *job submission manager* simulates the task arrivals corresponding to a user-defined task arrival rate and distribution function. The *resource information manager* maintains all sorts of information about the nodes. The details of the functionality of this subsystem are given in Chapter 4. In Section 5.4, we discuss the implementation details of the data structure which maintain the system information. This consists of static and dynamic information. The static information contains fixed reconfigurable area, hardware family or type, etc. The dynamic information includes the current set of processor *configurations*, the state (currently *idle* or *busy*), number of currently running tasks, available reconfigurable area etc. This information is required by various other modules, such as the *task scheduling manager or monitoring module* to perform different jobs.

The *core* subsystem: this subsystem is the core of the framework and it consists of a *task scheduling manager,* a *monitoring module,* and a *load balancing module*. The *task scheduling manager* can implement different scheduling policies to schedule tasks onto various nodes. It works closely with the *re-*

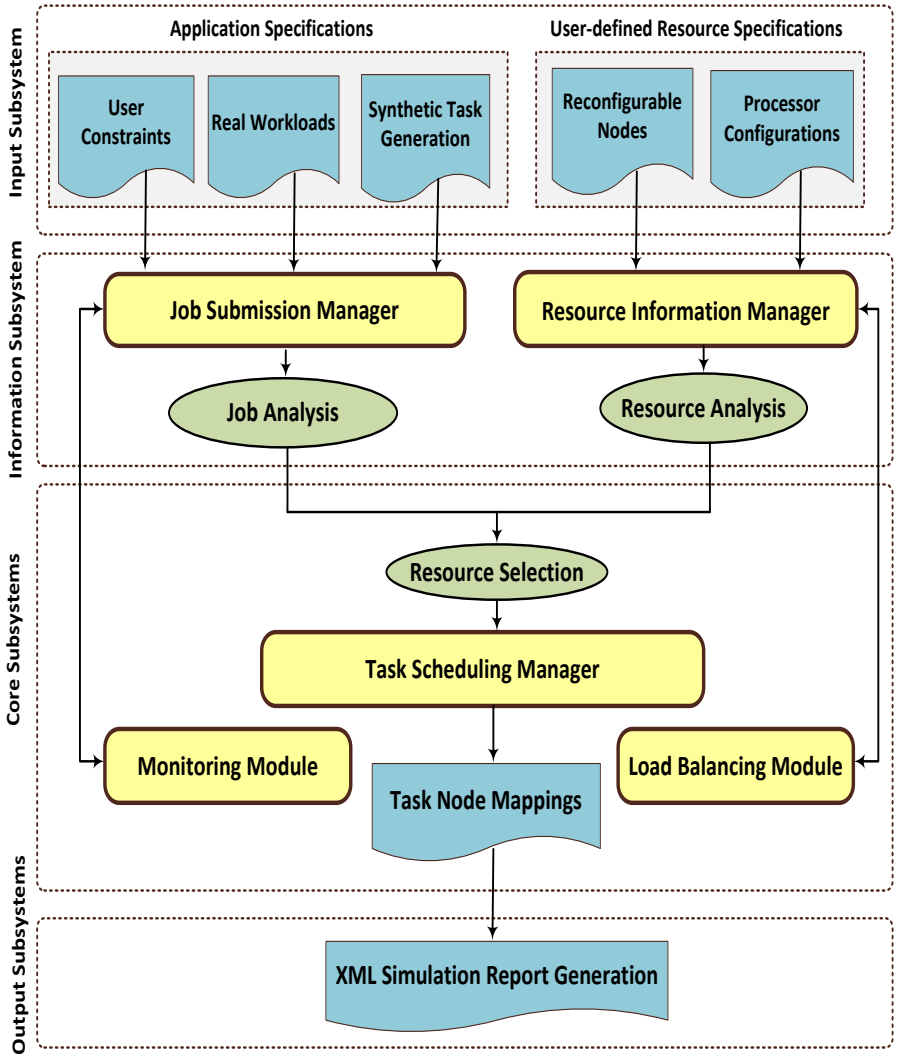


Figure 5.1: The top-level organization of DReAMSim. The *input subsystem* defines the input interface to the system. The *information subsystem* maintains the static and dynamic information in the system. The *core subsystem* mainly implements the task scheduling and load balancing modules. The *output subsystem* generates the simulation results.

source information manager in order to acquire updated statuses of the nodes, before the scheduling of the tasks. Due to the dynamic nature of a distributed system with reconfigurable nodes, the task scheduling process becomes a significant matter. First of all, as a result of the large number of nodes, it is hard to address issues such as load balancing when a task is scheduled to the available set of nodes. Secondly, since the nodes are reconfigurable and the system is dynamic, the scheduling should be adaptive. For these reasons, the framework contains a *load balancing module*. The current states of different nodes can be checked by the *monitoring module*.

The *output* subsystem: contains an XML *simulation report generator* which accumulates the statistics associated with various performance metrics that are produced by the framework and are gathered during each simulation run. These performance metrics are discussed in detail in Section 5.5.

5.3 Important Terminologies

In this section, we define some important terminologies used in the chapter. Table 4.1 enlists these terminologies briefly. Here, we describe them in more detail.

Reconfigurable node

Defined in Tuple 4.1, a node is a reconfigurable device with a fixed area. It can be reconfigured with a particular processor configuration, which can execute an application task. A node contains a limited reconfigurable area—defined in terms of slices or look-up tables (LUTs)—which is called $\mathit{TotalArea}$.

Configuration

Defined in Tuple 4.2, a configuration is a processor that can be configured on a node. It utilizes a particular reconfigurable area on the node, and it has a list of attributes required by a task.

Preferred configuration (C_{pref})

A preferred configuration (C_{pref}) is the processor configuration required by an application task. This configuration is a specific processor implemented on a reconfigurable node.

Closest configuration ($C_{closest}$)

A closest configuration ($C_{closest}$) is the closest possible configuration to the C_{pref} which can execute the task, but with some penalty. The criteria for $C_{closest}$ is set, based on some rules. For instance, one criteria can be that, its required reconfigurable area (*ReqArea* in Tuple 4.2) is minimum among all configurations with a required area more than that of C_{pref}.

Application task

An application task is defined in Tuple 4.3. It requires a specific processor configuration for its processing. The execution of a task takes a particular amount of time, termed as *Task Completion Time*.

Full reconfiguration

In this scenario, a node can accommodate only one configuration at a given time. In this way, only one task can be executed at one time. When the current task finishes its execution, then the node can be reconfigured with another configuration.

Partial reconfiguration

In this scenario, a node can be reconfigured for multiple configuration, simultaneously. In this manner, more than one tasks can run at a given time. Instead of reconfiguring the whole node, only a region can be utilized to add a new configuration.

Idle or *busy* node

Any node that contains at least one configuration, but it is currently executing no task(s) is termed as *idle* node. A *busy* node contains at least one configuration, and it is currently running at least one task.

Full node

A *full* node contains maximum number of configurations, and all of them are currently executing tasks. No further configurations can be added to the

node, because there is no more `AvailableArea`, which can be computed by Equation 4.4. It can be noticed that, in case of *full* reconfiguration scenario, any *busy* node is a full node, as it cannot currently accommodate any further new task.

Idle or *busy* list

Idle (*busy*) list is a linked list of *idle* (*busy*) nodes containing the same configuration. We explained the functionality of these lists in Chapter 4, by describing a detailed motivational example. It can be noticed that, in case of *partial* reconfiguration, each node can be part of multiple *idle* or *busy* lists.

Sus list

It is defined as the list of tasks that are currently suspended by the scheduler. At a certain time, if a task cannot be scheduled due to the unavailability of a node, it is put in suspension list.

Timetick

It is the basic unit of progression of time on a target system. All the metrics related to time in this thesis, are based on *timetick* unit of time.

Search step

A search step is the basic unit of exploration to search a memory location. It helps in computing the total number of scheduling steps required to assign a task to a node.

Scheduler workload

It is the total number of search steps required to assign a task to a node and to perform various housekeeping jobs, for instance updating the *idle* and *busy* lists.

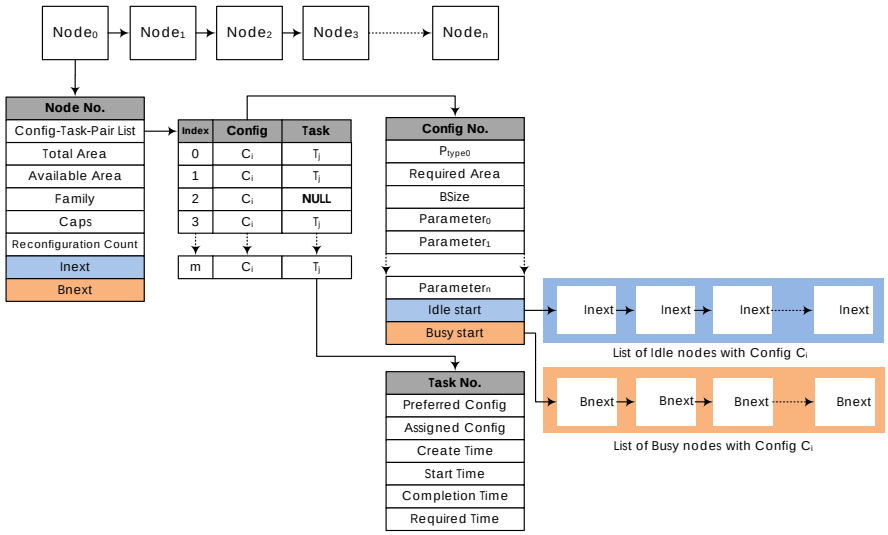


Figure 5.2: Dynamic data structures in the *resource information manager*. Each instance of node contains a *Config-Task-Pair* list, that maintains the current set of configurations and tasks on the node. A new *Config-Task* entry is created, when the node is reconfigured with a new configuration.

5.4 Design and Implementation of DReAMSim

In this section, we discuss the design and implementation details of the DReAMSim. We first present the details of the dynamic data structures for resource management, and their implementation; and then, we provide the simplified UML model of the framework.

5.4.1 Implementation of the Data Structures

We designed and implemented effective and dynamic data structures to provide a simple mechanism for the maintenance of information corresponding to dynamically reconfigurable nodes in the *resource information manager*. Figure 5.2 depicts the proposed data structure. Information regarding all reconfigurable nodes in the system is maintained by a data structure denoted by *node list*. Each item (node) in this list updates the information about a node and it contains a *config-task-pair list*, `TotalArea`, `AvailableArea`, and other node attributes. It also contains two pointers namely, *Inext* and *Bnext*. As depicted in Figure 5.3, these pointers are used to link all the *idle*

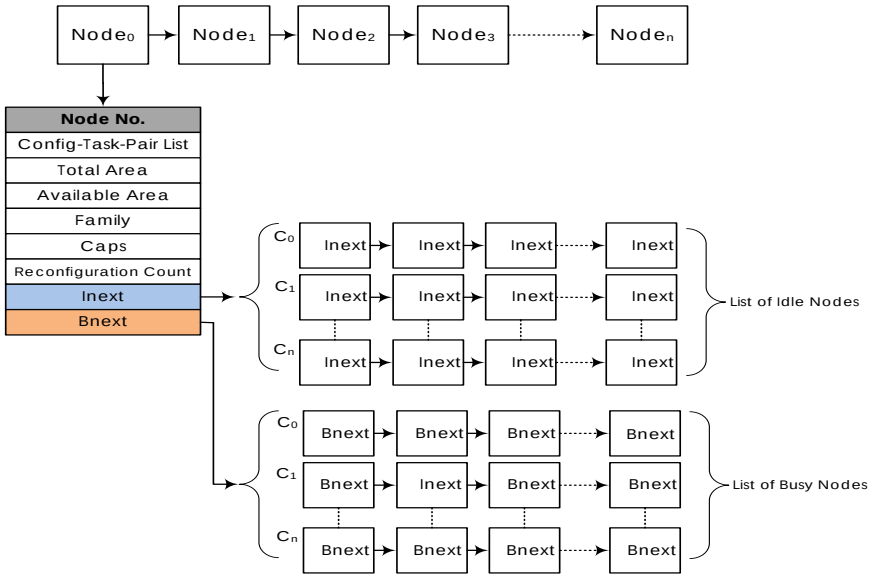


Figure 5.3: The pointer *Inext* (or *Bnext*) link a particular node to the next *idle* (or *busy*) node with the same configuration. Multiple lists of *idle* and *busy* nodes are simultaneously maintained.

or *busy* nodes of a certain configuration. This figure depicts the functionality of these pointers. In each case, *Inext* (and *Bnext*) points to the next *idle* (or *busy*) node with the same configuration. In this way, the *resource information manager* updates the lists of all *idle* and *busy* nodes with a certain configuration.

The *config-task-pair list* is another data structure which maintains the *configuration-task* pairs on a particular node. It keeps track of all the configurations on a node. Whenever a new configuration (C_i) is made on the node, a new *configuration-task* entry is created in the list. In this figure, all the actively running tasks are represented by T_j, where j is the ID of the task. If there is no currently running task on a particular configuration on the node, then it is represented by *NULL*. The maximum number of *configuration-task* pairs on a node depends on the `AvailableArea` of the node. For instance, if a new task wish to create a new configuration on the node, then the `ReqArea` of that configuration must be less than the current `AvailableArea` of the node.

Each *configuration* instantiates a typical processor configuration of type P_{type}, its `ReqArea`, and a list of its `param`. Additionally, it consists of two

pointers *Idle start* and *Busy start*. *Idle start* (or *Busy start*) points to the first node of the linked list of the *idle* (or *busy*) nodes configured with this particular *configuration*. Chapter 4 provides a detailed description of how these pointers link various nodes during the resource management process.

The main reason to propose these pointers and linked lists is to maintain the dynamic behavior of the nodes which are expected to contain various configurations at different times during a simulation. In addition, these linked lists ease up the search effort needed to get the state information of a certain node. This search effort can become especially time-consuming, if the total number of nodes is very large.

5.4.2 UML Model of DReAMSim

A class diagram of the DReAMSim framework, represented by using simple, **U**nified **M**odeling **L**anguage (UML) notation, is depicted in Figure 5.4. It provides details of different classes implemented in order to generate tasks, nodes, configurations, scheduler, lists, and the simulation environment. The implementation of the DReAMSim is described in C++ language. The framework implements the following important classes:

- Node: implements a typical reconfigurable node and its capability is defined by `TotalArea`, `AvailableArea`, `family`, and `caps`. It performs all the basic functionalities that are required to instantiate a node. When a user inputs the parameters to start a simulation, this class initializes all the nodes and allocates them the user-defined area. Moreover, it adjusts the `AvailableArea` on a node, when a new configuration is installed on it. Similarly, it adds (or removes) tasks to a node. Some important methods of this class are described in the following:
  - *InitNodes():* initializes the number of nodes defined by the user, and allots a fixed `TotalArea` to each node within the upper (*NodeUpperA*) and lower (*NodeLowerA*) area limits.
  - *SendBitstream():* adds a configuration on the node, adjusts the `AvailableArea` according to the `ReqArea` of the new configuration, and increases the *reconfiguration count* on the node.
  - *MakeNodeBlank():* removes all the configurations on a particular node, and equates the `AvailableArea` to the `TotalArea`.

- *MakeNodePartiallyBlank():* removes one or more existing configurations on a particular node, and re-adjusts the `AvailableArea`.
- *AddTaskToNode():* adds a task to a particular node, if it contains the required configuration of the task.
- *RemoveTaskFromNode():* removes a task from a particular node.

The class contains several data members. For instance, *AvailableArea* and *ConfigCount* are dynamic data members of the class, and they are updated when the new configurations are installed on a node. Similarly, *NetworkDelay* defines the communication time (t_{comm}) required to send the task to the node from the scheduler. Furthermore, *INext* and *BNext* are two pointers that point to a node to the next node with the same configuration.

- Task: this is used to represent an application task, and is characterized by $t_{required}$, C_{pref}, and `data`. This class instantiates a typical task, sends it to a particular node, and calculates its related statistics. Once a task finishes its execution, it invokes a task completion procedure to perform various housekeeping jobs. Furthermore, this class helps to search for C_{pref} and $C_{ClosestMatch}$ of a given task. It contains the following methods:

  - *CreateTask():* creates a new task and its attributes, such as t_{create}, $t_{required}$, and its C_{pref}.
  - *SendTaskToNode():* sends a task to a particular node, and calculates its *start time (t_{start}), completion time ($t_{completion}$), and waiting time (t_{wait}).*
  - *TaskCompletionProc():* it is invoked by the scheduler, after a task finishes executing on a particular node. It releases the node to make it available for succeeding tasks. It also updates the corresponding *idle* and *busy* lists. Additionally, it reports some statistics after the completion of the task.
  - *FindPreferredConfig():* searches for the C_{pref} of a particular task among all the configurations in the *configurations list*. Currently, a simple linear search is employed but it can be made more intelligent.

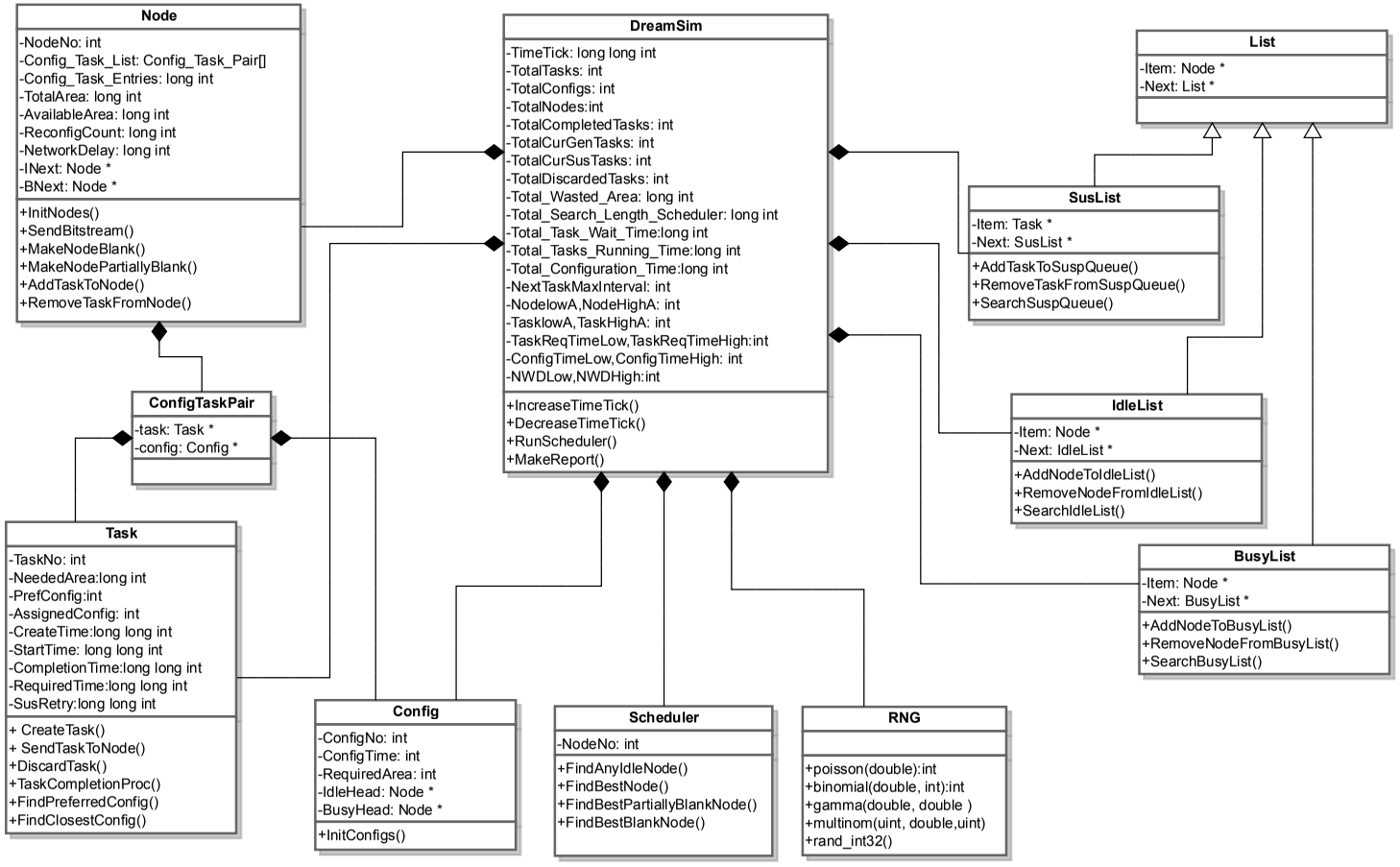


Figure 5.4: The UML model of DReAMSim framework. *SusList*, *IdleList*, and *BusyList* are the derived classes from the base class *List*.

  - *FindClosestConfig():* searches for the $C_{ClosestMatch}$ of a particular task if its C_{pref} is not available in the *configurations list*. The criteria for the $C_{ClosestMatch}$ is that its `ReqArea` is the minimum among all configurations with a `ReqArea` more than the `ReqArea` of the C_{pref}.

  The data member *PrefConfig* instantiates the ID of the preferred configuration (C_{pref}) of the task, whereas, the *AssignedConfig* is the ID of the configuration to which the task is actually assigned. In most cases, it is the same as C_{pref}, but in some cases, if the scheduler assigns a $C_{ClosestMatch}$ to the task, it is different from the C_{pref}. Moreover, the data member *NeededArea* defines the area required by the C_{pref} of the task.

- Config: this class represents a typical configurations with attributes such as, its `ReqArea`, P_{type}, `parameters`, `BSize`, and `ConfigTime`. When a user defines all the required parameters, this class initializes the list of configurations by assigning them the required area (`ReqArea`). It consists of the following method:

  - *InitiConfig():* initializes the *configurations list* and assigns `ReqArea` and `ConfigTime` to all the configurations within a user-defined range.

  The two data members *IdleHead* and *BusyHead* are two pointers that maintain the address of the first node of the linked-list of the nodes of this particular configuration.

- ConfigTaskPair: it is a dynamic data structure which is used to keep track of all tasks and their corresponding configurations on a certain node.

- IdleList and BusyList: these classes maintain linked lists for managing information corresponding to *idle* (or *busy*) nodes with a particular configuration. If a task finishes execution on a certain node, then it is added to the *idle* list of nodes (and removed from the *busy* list of nodes) with the configuration required by the task. These classes are derived from the *base* class *List*, which is responsible for maintaining all the lists in the system. They contain the following important methods:

  - *AddNodeToIdleList()* and *AddNodeToBusyList():* adds a node to the *idle* (or *busy*) list of nodes with a particular configuration.

  - *RemoveNodeFromIdleList()* and *RemoveNodeFromBusyList():* removes a node from the *idle* (or *busy*) list of nodes with a particular configuration.
  - *SearchIdleList()* and *SearchBusyList():* used to traverse the *idle* (or *busy*) list of a particular configuration.

- SusList: this class implements the suspension queue data structure to hold suspended tasks and contains the following methods:
  - *AddTaskToSusQueue():* implements a simple queue data structure which holds the tasks, put in suspension by the scheduler.
  - *RemoveTaskFromSusQueue():* each time a node finishes executing a task, the suspension queue is checked using this method to determine if a suitable task is waiting in the queue which can be executed on the node. It removes the task from the suspension queue and sends it the node.
  - *SearchSusQueue():* traverses the suspension queue to search a particular task.

- Scheduler: assists in various task scheduling policies defined by the user, and implements methods that perform important relevant jobs. It contains the following methods:
  - *FindBestNode()*: it is invoked to search the *best-node* match for a given task, among all the idle nodes configured with its C_{pref}. The criteria for the *best-match* depends on the scheduling strategies. For instance, *best-match* can be the node which possesses the minimum `AvailableArea`, so that the nodes with larger area can be utilized for future tasks which may require more reconfigurable area.
  - *FindAnyIdleNode():* it searches for any *idle* nodes configured with a configuration other than the C_{pref} of the task. It is explained in Algorithm 6.2.

- DreamSim: it is the core class of the simulator and it interacts with other classes to prepare the system to run the user-defined simulations. Two important methods of this class are given in the following:
  - *RunScheduler():* it simulates a certain task scheduling policy implemented in the scheduler.

Class	Description	Important methods
Node	Implements a typical reconfigurable node and related functionalities. It initializes the nodes, adds tasks to the nodes, and adds configurations to a node.	*InitNode(), SendBitstream(), MakeNodeBlank(), AddTaskToNode(), MakeNodePartially-Blank(), RemoveTaskToNode().*
Task	Creates task instances, and related functionalities. It sends a task to a node, interacts with scheduler after task completion, and updates the statistics.	*CreatTask(), SendTaskToNode(), DiscardTask(), TaskCompletionProc(), FindPreferredConfig(), FindClosestConfig().*
Config	Implements a processor configuration and its attributes.	*InitConfigs()*
ConfigTaskPair	Keeps track of the currently running tasks and their corresponding configurations on a given node.	—
Scheduler	Implements various task scheduling strategies.	*FindAnyIdleNode(), FindBestNode(), FindBestPartiallyBlan-kNode(), FindBestBlankNode().*
DreamSim	Implements the simulator and interacts with other classes to run user-defined simulations.	*IncreaseTimeTick(), DecreaseTimeTick(), MakeReport().*
List	Maintains various lists during a simulation run.	—
IdleList/BusyList	Maintain the idle and busy lists of nodes.	*AddNodeToIdleList(), RemoveNodeFromBu-syList(), SearchIdle().*
RNG	Generates random numbers to be used in various parameters in the simulation.	*Poisson(), Binomial(), Gamma().*

Table 5.1: Important classes and methods of DReAMSim framework.

  - *MakeReport():* accumulates the statistical data during the simulations.
  - *IncreaseTimetick*() and *DecreaseTimetick*(): these methods simulate the progress of time in the simulator.

  In this class, most of the data members are required to input the parameters defined by the user. They help to initialize the node area limits, configuration area bounds, required time by tasks, reconfiguration time limits, and network delay limits, etc.

- RNG: it is a random number generator class which is based on the *Ziggurat Method* [122] using the algorithm described in [123] for generating *Gamma variables*. It provides several random number distributions, such as *Poisson*, *Binomial*, *Gamma*, *Uniform random*, etc. The simulator uses these random number distributions to generate configurations, nodes, and task arrivals.

In each class, various methods utilize *data members* in an appropriate manner. For instance, *NextTaskMaxInterval* defines the upper time limit during which a new task is generated during a particular simulation run.

The *worst case time complexity* of running simulations on DReAMSim is $O\left(nT^2 + nT.O\left(algorithm\right)\right)$, where nT is the number of tasks and $O(algorithm)$ is the runtime of the task scheduling algorithm implementation.

5.5 Performance Metrics Generated by DReAMSim

The method *MakeReport()* in the DReAMSim class accumulates the statistical data at the end of each simulation. Table 5.2 provides some important statistics generated by the simulator. *Total scheduler workload* is the sum of *search steps* taken by the simulator to schedule a task and to do different house keeping activities, for instance, updating the idle, busy, and suspension queue lists. A *search step* is a basic unit of exploration to search a memory location. A scheduling step counter *SL* is incremented after each search step that the *scheduler* takes to schedule a task. In the following, we explain how these metrics are computed.

Performance metric	Description
Average wasted area per task	Average reconfigurable area wasted during the scheduling of all tasks during a simulation run. See Equation 5.4.
Total task completion time	Total time lapse from the arrival of the task to the system until its completion. See Equation 5.9.
Average reconfiguration count per node	Average number of reconfigurations performed by each node during the simulation.
Average reconfiguration time per task	Average reconfiguration time required per task during the simulation. See Equation 5.8.
Average waiting time per task	Average time elapsed from the time a task is submitted to the system until the time it is assigned to a node. See Equation 5.6.
Average scheduling steps per task	Total number of search links explored by the scheduling system to assign a task to a proper node. This metric closely reflects the quantitative value of the time taken by the scheduling system to accommodate a task.
Total discarded tasks	Total number of tasks discarded during the simulation.
Total scheduler workload	Total number of *search steps* taken by the *resource information manager* to perform various housekeeping jobs such as, maintaining the current states of nodes and configurations.
Total nodes utilized	Total number of nodes utilized during a simulation run.
Total simulation time	Total simulation time required to execute all tasks during the simulation. See Equation 5.1.

Table 5.2: Some important DReAMSim performance metrics.

5.5.1 Total simulation time

It is the *total simulation time* needed to execute all tasks during a given simulation run. In class *DreamSim*, the methods *IncreaseTimeTick()* and *DecreaseTimeTick()* simulate the progress of time in terms of *timeticks* in a general manner, where *timetick* is a unit of time on a target system. In this respect, the total simulation time is calculated as follows:

$$\mathit{Total\ simulation\ time} = \mathit{Total\ number\ of\ timeticks} \tag{5.1}$$

5.5.2 Average wasted area per task

It is the average reconfigurable area wasted during the scheduling of all tasks during a simulation run. It is calculated as the sum of `AvailableArea` of those nodes, which contain at least one configuration during a simulation run. It is because the `AvailableArea` is the leftover reconfigurable area on a node after a reconfiguration is performed. Consequently, it can be considered as *wasted*, as it is not utilized for any purpose until another reconfiguration is performed.

Based on Tuples 4.1, 4.2, and 4.3, the `AvailableArea` of a node can be computed as follows:

$$\mathit{AvailableArea} = \begin{cases} \mathit{TotalArea} & \text{if } k = 0 \\ \mathit{TotalArea} - \sum_{j=1}^{k} \mathit{ReqArea}_j & \text{if } k > 0 \end{cases} \tag{5.2}$$

Where k is the total number of current configurations of the node. Subsequently, the *total wasted area* at any given time is calculated as follows:

$$\mathit{Total\ wasted\ area} = \sum_{i=1}^{N'} \mathit{AvailableArea}_i \tag{5.3}$$

Where N' represents the total number of those nodes which contain at least one configuration. Thus, the *average wasted area per task* is evaluated as given:

$$\mathit{Avg\ wasted\ area\ per\ task} = \mathit{Total\ wasted\ area} / \mathit{Total\ tasks} \tag{5.4}$$

It is expected that for a given set of parameters, if a node allows multiple configurations, then the *average wasted area per task* will be less as compared to the scenario, if the node allows only one configuration at a given time.

5.5.3 Average waiting time per task

It is the average time elapsed from the time a task is submitted to the system until the time it is scheduled to a node. It includes the configuration time to set up the required configuration on the target node, and the communication time to send the task. The total waiting time for each task (t_{wait}) is calculated as follows:

$$t_{wait} = t_{start} - t_{create} + t_{comm} + t_{reconfig} \tag{5.5}$$

Where t_{create} is the time when the task is created, t_{start} is the time when the task is submitted to a node for execution, t_{comm} is the communication time of the task to reach the node, and $t_{reconfig}$ is the time to reconfigure the node, if any configuration is required before task allocation. Similarly, the *average waiting time per task* is computed as follows:

$$Avg\ waiting\ time\ per\ task = \sum_{j=1}^{T} (t_{wait})_j / Total\ tasks \tag{5.6}$$

Where T represents the total number of tasks executed during a simulation.

5.5.4 Average reconfiguration count per node

It is the average number of reconfigurations performed by each node during the simulation. When a new configuration is added to a particular node, its reconfiguration count is increased by 1. An average reconfiguration count per node can be computed as follows:

$$Total\ reconfig\ count = \sum_{k=1}^{N} (ReconfigCount)_k / Total\ nodes \tag{5.7}$$

Where N represents all those nodes that have gone through at least one reconfiguration.

5.5.5 Average reconfiguration time

It is the time required to reconfigure a new configuration on a node, during the simulation. We calculate the *total configuration time* as follows:

$$Total\ reconfig\ time = \sum_{k=1}^{C} (ReconfigCount)_k . (ReconfigTime)_k \tag{5.8}$$

Where C is the total number of configurations in the system, and the term $\sum_{k=1}^{C} (ReconfigCount)_k$ computes the *total reconfiguration count* by all the nodes.

5.5.6 Average scheduling steps per task

It is the total number of search links explored by the scheduling module to assign a task to a proper node. For instance, the scheduler explores various nodes to find an *idle* node or to find the *best-match* node. When the corresponding methods are invoked to perform such exploration, the scheduling step counter increase accordingly. This metric closely quantifies the time taken by the scheduler to accommodate a task.

5.5.7 Total scheduler workload

It is the total number of *search steps* taken by the *resource information manager* to perform various housekeeping jobs during one simulation run. These jobs include the dynamic maintenance of *idle* and *busy* lists, and the management of the current states of nodes and configurations.

5.5.8 Total task completion time

It is the time lapse from the arrival of the task to the system until its completion. The total task completion time ($t_{completion}$) includes the total waiting time for the task (t_{wait}) and the time required ($t_{required}$) to complete its execution. It can be computed as follows:

$$t_{completion} = t_{wait} + t_{required} \tag{5.9}$$

5.5.9 Total nodes utilized

It is the total number of nodes utilized during a simulation run. This metric highly depends upon the scheduling strategy and other parameters in the system.

5.5.10 Total discarded tasks

It is the total number of tasks discarded during the simulation. There can be a scenario where a task requires a certain configuration which cannot be reconfigured on any node in the system due to its `ReqArea` that is larger than the area of each node in the system. In this case, the task is discarded. This metric can be utilized to test the functionality of the simulation framework.

5.6 Task Scheduling Algorithm — A Case Study

In order to test the framework, we implemented a simple dynamic scheduling algorithm as a case study. In this section, we provide the details of this algorithm, which supports reconfigurable nodes. For simplicity, we allow only one configuration per node at a given time. In this case, once a certain task is finished on a node, it becomes idle, then the scheduler can reconfigure another configuration. In Chapter 6, we discuss a scenario where a node can accommodate multiple configurations simultaneously. Code listing 5.1 provides an overview of the scheduling algorithm. It takes tasks, configurations, and nodes as inputs and generates task-node mappings. These mapping decisions are taken based on the scheduling process, which is explained as follows.

At each scheduling step, an incoming task (current task or *CT*) is assumed to have its preferred configuration (C_{pref}) as shown in Tuple 4.1. If possible, the algorithm allocates the *CT* onto a node that is already reconfigured with its C_{pref}. Firstly, the C_{pref} of the *CT* is explored in the *configurations list* depicted in Figures 5.2 and 5.3. If the C_{pref} is found in this list, then the algorithm searches for idle node(s) of that configuration by traversing the list of idle nodes of that specific configuration. If idle nodes are available, the *CT* is assigned to the *best-match* node. Here the criteria for the *best-match* node is the one that wastes minimum amount of reconfiguration area among all the nodes in the linked list. In other words, it is the node that has minimum `AvailableArea` after the C_{pref} is configured on it. In a scenario

Listing 5.1: A simple dynamic scheduling algorithm supporting reconfigurable nodes.

```
1  Initialize the configs list
2  Initialize the nodes list
3  Begin TaskSchedule (Task CT)
4  if (Exact match of Cpref of CT is available in configs list)
5        if (Idle node(s) with exact match available in system)
6              if (best match with sufficient area available)
7                    Allocate CT
8                    Update the corresponding idle and busy lists
9                    END
10             else
11                   goto A
12       else
13 A:           if (Any blank node(s) available)
14                   if (Bestmatch with sufficient area available)
15                         Reconfigure the blank node with Cpref of CT
16                         Allocate CT
17                         Update the blank and corresponding busy lists
18                         END
19                   else
20                         goto B
21             else
22 B:                 if (Any idle node(s) with sufficient area available)
23                         Reconfigure selected node with Cpref of CT
24                         Allocate CT
25                         Update the corresponding busy/idle lists
26                         END
27                   else
28                         goto C
29 else
30 C:    if (Closest match of Cpref of CT available)
31                   if (Any idle node(s) with closest match available)
32                         if (Bestmatch for closest match available)
33                               Allocate CT
34                               Update the corresponding busy/idle lists
35                               END
36                         else
37                               goto D
38                   else
39                         goto D
40 D:           if (CT required exact match of Cpref initially )
41                         if (Any blank node(s) with sufficient area)
42                               if (Bestmatch blank node available)
43                                     Reconfigure the node
44                                     Allocate CT
45                                     Update the busy/idle lists
46                                     END
47                               else
48                                     goto E
49                         else
50                               goto E
51             else
52                   goto E
53 E:    if (Atleast one busy node available with sufficient area)
54                   Put CT in suspension queue
55                   END
56       else
57             Discard CT
58             END
```

when the C_{pref} is found in the *configurations list*, but idle nodes are not available, then the list of *blank* nodes is traversed to accommodate *CT* (see line 13 in the code listing 5.1). *Blank* nodes are currently not configured with any configuration. In this case, C_{pref} is reconfigured on the *blank* node, and *CT* is assigned to that node. If neither *idle* nor *blank* nodes are found with the configuration C_{pref}, then the algorithm explores any *idle* node(s) that are reconfigured with any configuration that is different than C_{pref} of the *CT*, by searching the *configurations list* (see line 22 in the code listing 5.1). Consequently, a node with a minimum sufficient area is chosen, reconfigured with the C_{pref} of the *CT*, and the task is allocated to it.

If the C_{pref} of the *CT* is not available in the *configurations list* (see line 30 in the code listing 5.1), the algorithm searches for a node with the *closest-match* configuration ($C_{ClosestMatch}$) to C_{pref} of *CT*, in the *configurations list*. Once the $C_{ClosestMatch}$ is found, the algorithm searches for *idle* nodes with that configuration. Otherwise, it searches for *blank* nodes with sufficient area to assign the *CT*. If *blank* nodes are also not found and at least one *busy* node is available with sufficient area (see line 53 in the code listing 5.1), then the algorithm puts the task in a suspension queue to wait for any *busy* node of the required configuration to become *idle*. Finally, if no node with sufficient area can be used to accommodate the task, it is discarded. After the allocation process of the *CT*, the *busy* or *idle* lists of nodes of the corresponding configurations (C_{pref} or $C_{ClosestMatch}$) are updated and maintained accordingly.

5.7 Simulation Environment and Results

In the *task scheduling manager*, we implemented the algorithm that provides the support for reconfigurable nodes, as explained in Section 5.6. Subsequently, we conducted a number of experiments with various simulation parameters and computed a set of performance metrics (Table 5.2). The experimental results exhibit expected trends. In the remainder of this section, we elaborate on the simulation platform, simulation parameters, and discuss the experimental results in detail.

Simulation platform: The experiments were performed on Intel Core 2 Duo CPU E8400, 64-bit machine, running openSUSE 11.3 with the Linux kernel 2.6.34. The processor runs at 3.00GHz. The implementation of the simulation framework was described in C++ language and compiled using gcc v.4.5.0 for the above-mentioned target processor. On this experimental setup, a typical simulation run for 10,000 tasks with 100 nodes and 50 configurations

Parameter	Value
Total tasks generated	1000...100000
Total nodes	8, 16, 64, 100, 200, 300, 500, 1000
Total configurations	10, 20, 30, 40, 50
Tasks generation interval	[1...50] and [100...1000]
Configurations required area (*RegArea*) range	[100...2500]
Node total area (*TotalArea*) range	[1000...5000]
Task required timeslice ($t_{required}$) range	[100...10000]
Reconfiguration time ($t_{reconfig}$) range	[1...5]
$C_{ClosestMatch}$ percentage	10 %
Reconfiguration method	*without partial configuration*

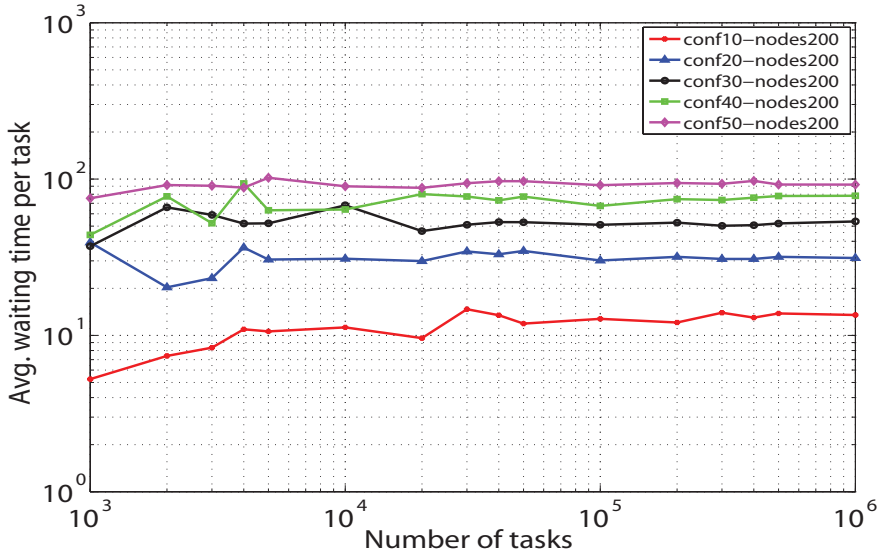
Table 5.3: Various simulation parameters and their values. The time ranges are in *timeticks*, and the area ranges quantify typical area units, e.g., *slices* or *look-up tables* (LUTs).

requires approximately 39 seconds.

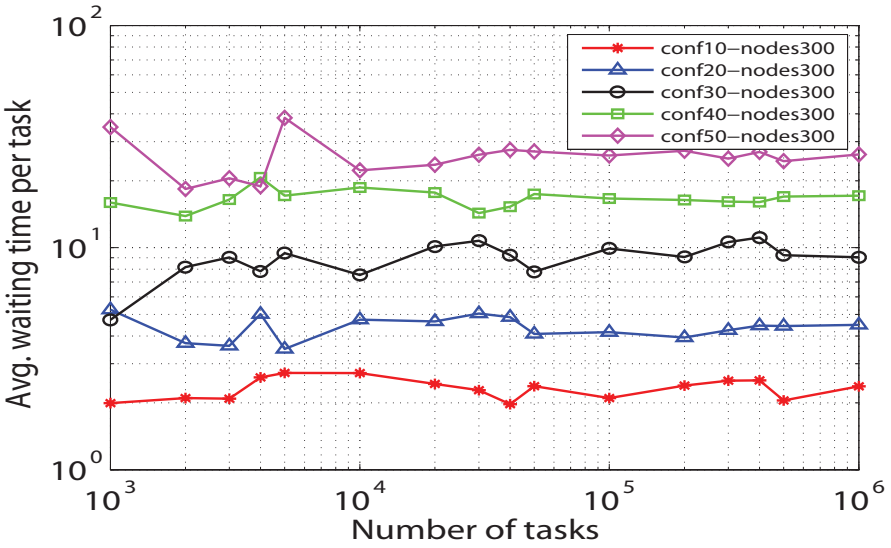
Simulation parameters: The simulation parameters used in our experiments are presented in Table 5.3. We conducted two sets of simulation experiments. The task arrival rate for the 1st set of experiments is in the interval of [1..50] timeticks with uniform distribution, whereas, the task arrival rate is set to [100...1000] timeticks in the 2nd set of experiments. The total number of tasks in each experiment varies between [10^3...10^6] and their completion time varies randomly between [100...10000] timeticks. Similarly, the preferred configuration (C_{pref}) for any given task requires area within range between 100 to 2500 area units (e.g., area slices or FPGA LUTs). The available reconfigurable hardware area range is set between 1000 to 5000 area units. For 10 % of the total tasks, we assign a preferred configuration (C_{pref}) that can not be found in *configurations list*. Therefore, these tasks are assigned to the reconfigurable nodes with *closest-match* configuration ($C_{ClosestMatch}$) by the scheduling algorithm at run time. Finally, the reconfiguration time for any reconfigurable node in the distributed system varies between [1...5] timeticks.

5.7.1 Results Discussion

In this subsection, we present and discuss the simulation results for some of the important performance metrics such as, *average waiting time per task*, the

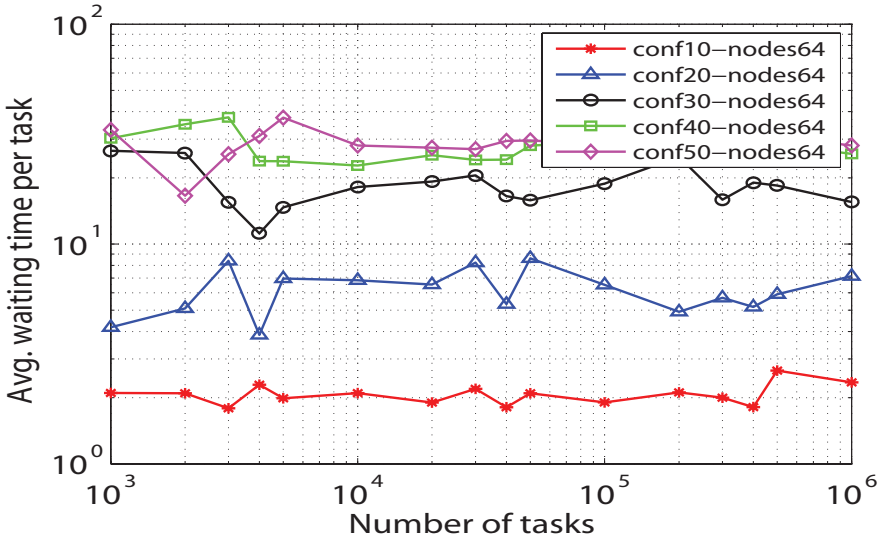


(a) Total nodes=200 (y-axis on log-scale).

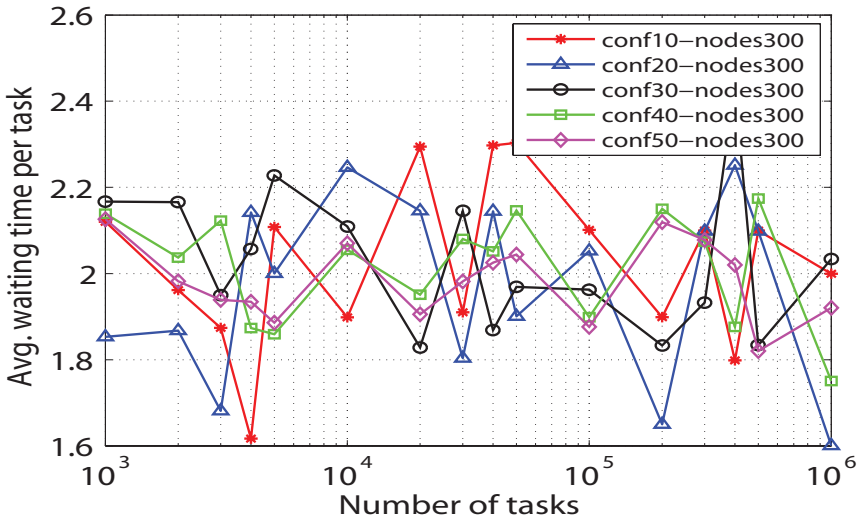


(b) Total nodes=300 (y-axis on log-scale).

Figure 5.5: The *average waiting time per task* for variable number of nodes and fixed sets of configurations (10, 20, 30, 40, 50). Next task arrival within the interval [1...50].



(a) Total nodes=64 (y-axis on log-scale).



(b) Total nodes=300.

Figure 5.6: The *average waiting time per task* for variable number of nodes and fixed sets of configurations (10, 20, 30, 40, 50). Next task arrival within the interval [100...10000].

average scheduling steps required per task, and *average reconfiguration count per node*.

The average waiting time per task: The *average waiting time per task* against given sets of tasks, configurations and nodes are depicted in Figure 5.5. The task arrival rate is set as [1...50] with uniform distribution.

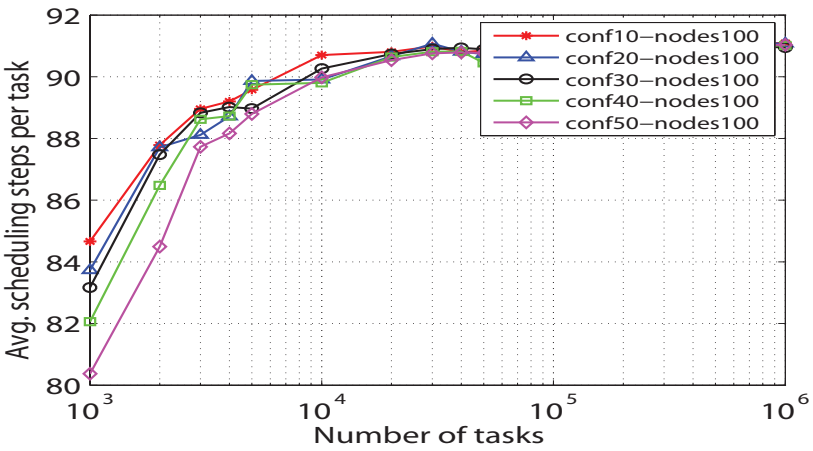
Figure 5.5a depicts the case when the total number of nodes is 200. It can be noticed that the quantitative *number of average waiting time per task* increases for larger set of possible configurations (e.g., 40 or 50) and vice versa. This is because there are more nodes/configuration available to accommodate an incoming task in case of less number of configurations. Similarly, for 300 nodes, depicted in Figure 5.5b, the *average waiting time per task* decreases as compared to the case with 200 nodes. For instance, the *average waiting time per task* for different configurations and 200 nodes varies between 10 to 100 timeticks and it varies 2 to 40 timeticks for 300 nodes.

In Figure 5.6, when the task arrival interval is changed to [100...1000], the *average waiting time per task* decreases considerably for each case of 64 and 300 nodes. It can be noticed in Figure 5.6a that the *average waiting time per task* ranges between around 1 to 30 for only 64 nodes, which is similar to Figure 5.5b which has a lot more number of nodes (300 as compared to 64 nodes). Figure 5.6b depicts that for 300 nodes with arrival rate interval [100...1000], the *average waiting time per task* decreases significantly at around 2 timeticks.

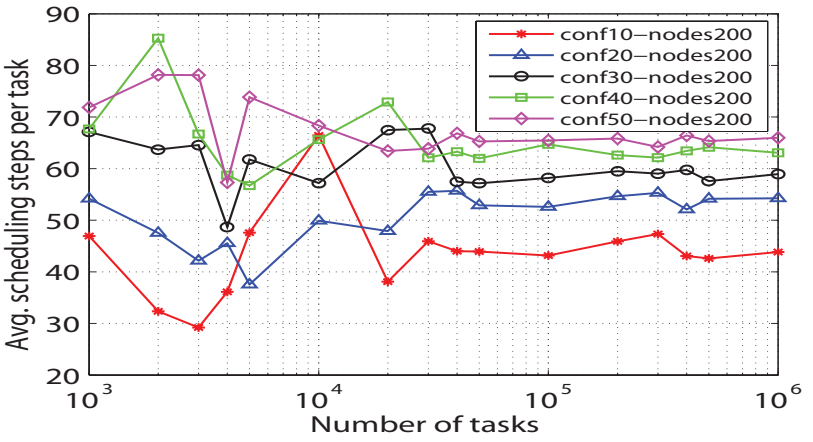
Overall, it can be concluded that for a given task arrival interval, the *average waiting time per task*, decrease as the number of nodes increase in the system. Secondly, an increase in the task generation interval results in a decrease in the *average waiting time per task*. These results correspond to typical trends of the *average waiting time per task* in a distributed system for a given set of configurations and nodes.

The average scheduling steps per task: The *average number of scheduling steps per task* against given sets of tasks, for fixed set of configurations (10, 20, 30, 40, and 50) and different number of nodes, are depicted in Figures 5.7 and 5.8. Figure 5.7 depicts the case when the next task generation is set as [1...50] in uniform distribution. The number of nodes are set to 100, 200 and 300. Similarly, 5.8 depicts the case when the next task generation interval is set to [100...1000] and number of nodes are set to 64, 100, 300.

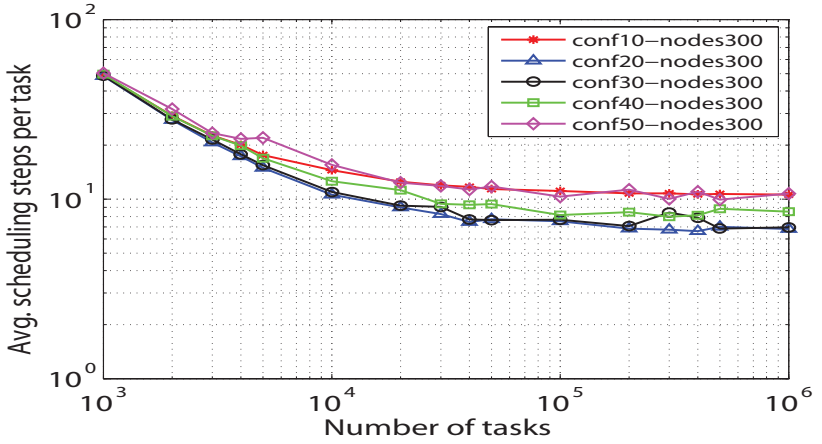
Figure 5.7 (a) shows that, in case of 100 nodes, an increase is observed in the *average scheduling steps per task*, for a specific range of tasks (between 1000



(a) Total nodes = 100.



(b) Total nodes = 200.



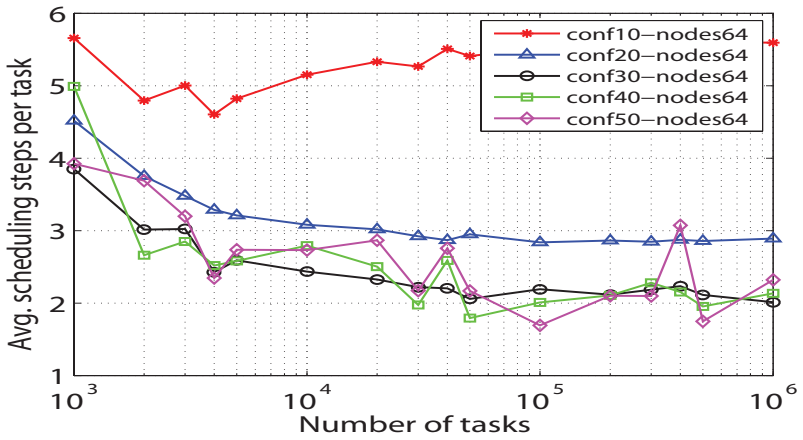
(c) Total nodes = 300.

Figure 5.7: The *average scheduling steps per task* for fixed sets of configurations (10, 20, 30, 40, 50), and next task arrival interval=[1...50]. (a) Total nodes = 100, (b) Total nodes = 200 and (c) Total nodes = 300.

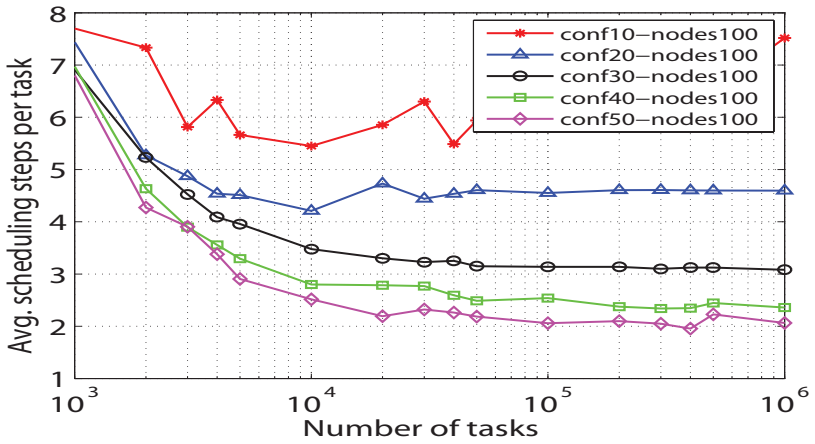
and $10,000$). It is because the system cannot accommodate the tasks in a timely manner due to a small number of nodes and they are frequently put in the suspension queue. This imposes additional search burden on the scheduler during the simulation process. For a relatively large number of tasks (10,000 and up), the average scheduling steps curve tends to be more stable, indicating that the system response is predictable for higher number of tasks. The curves show that the arrival pattern and rates of the incoming tasks are the same which increases the system response time but the *average scheduling steps per task* remain the same. Additionally, it should be noted that the *average scheduling steps per task* are higher (around 80 to 92) as compared to a scenario with higher number of nodes where *average scheduling steps per tasks* are lower (approximately $50 - 70$ for 200 nodes and 10 for 300 nodes).

Figure 5.7 (b) depicts that, for a specific range (between 1000 and $10,000$) of tasks, the curves have a bouncing effect which shows that the resources (200 nodes) are still inadequate to accommodate the tasks. The curves become smoother for larger number of tasks (10^4 and above) to show that the corresponding lists of nodes are formed to accommodate each task to its required preferred node. The system response becomes more predictable and the bouncing effect of lower number of tasks is alleviated due to averaging them on large number of tasks. It can also be noticed that the *average number of scheduling steps per task* are in the range between 40 and 80 steps per task.

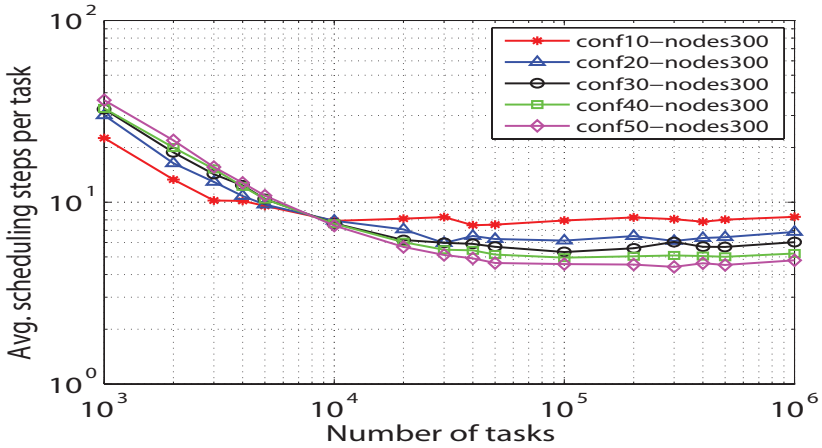
Figure 5.7 (c) depicts the *average scheduling steps per task*, when the nodes are increased to 300. It can be observed that the system response is more stable for higher number of tasks (10^4 and above) with quantitative number of steps per task around 10 steps per task. This case shows that there are enough resources (300 nodes) available to accommodate the incoming tasks. Initially, the number of steps are higher ($10 - 50$) depicting that the scheduler has to search *blank* node list of 300 nodes to find a suitable (*best-match*) node for the tasks. However, after 10^4 tasks, the behavior of the scheduler is stable at around 10 steps per task, indicating that the configuration lists are formed on the nodes and not much nodes are left as *blank*. Once the nodes are removed from the *blank* node list, the nodes are set up with different configurations and an incoming task gets more probability to find nodes with its C_{pref}, decreasing search burden on the scheduler. On average, for 30 configurations, the scheduler has to search for $300/30 \backsimeq 10$ *nodes per configuration* to accommodate an incoming task, thus confirming the *average scheduling steps per task* around 10 in the steady-state situation. Figures 5.7 (a), (b), and (c) indicate that an increase in resources results in a more stable system response and decrease in *average scheduling steps per task*.



(a) Total nodes = 64.



(b) Total nodes = 100.



(c) Total nodes = 300.

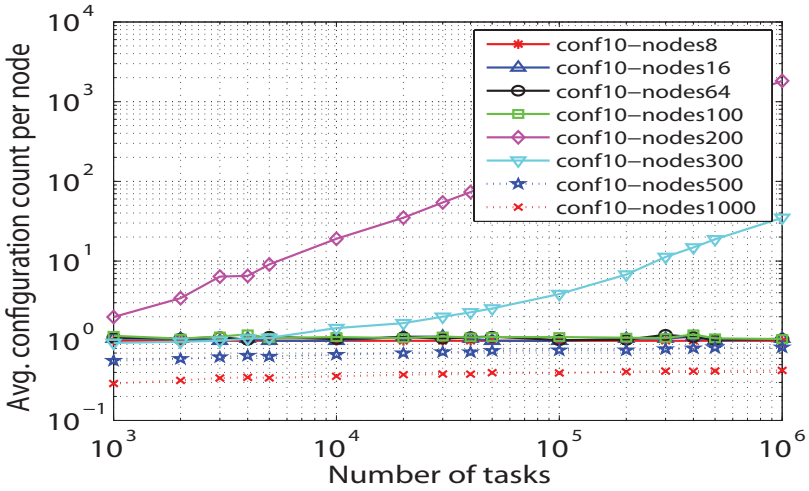
Figure 5.8: The *average scheduling steps per task* for fixed sets of configurations (10, 20, 30, 40, 50), and next task arrival interval=[100...1000]. (a) Total nodes = 64, (b) Total nodes = 100 and (c) Total nodes = 300.

Subsequently, we discuss the case depicted in Figure 5.8 when the task generation interval is changed to [100...1000]. Figure 5.8 (a) depicts that the scheduler has more time to accommodate an arriving task as the tasks are not arriving rapidly now. The bouncing effect can still be observed due to less number of nodes (64). The quantitative number of steps is around $2-6$ steps per task, which is very low showing less burden on the scheduler due to tasks arriving in relatively slow manner. Figures 5.8 (b) and (c) depict that the resources are more or less sufficient to accommodate an arriving task. Figure 5.8 (c) shows that the behavior of the curves is more stable but the number of steps is a bit higher (between $10-40$) initially because the number of nodes are more than sufficient. This indicates that the scheduler searches for 300 nodes in the *blank* nodes lists to accommodate an arriving task, thus the number of steps is higher in stable region (for 10,000 and above tasks) as compared to 5.8 (a) and (b).

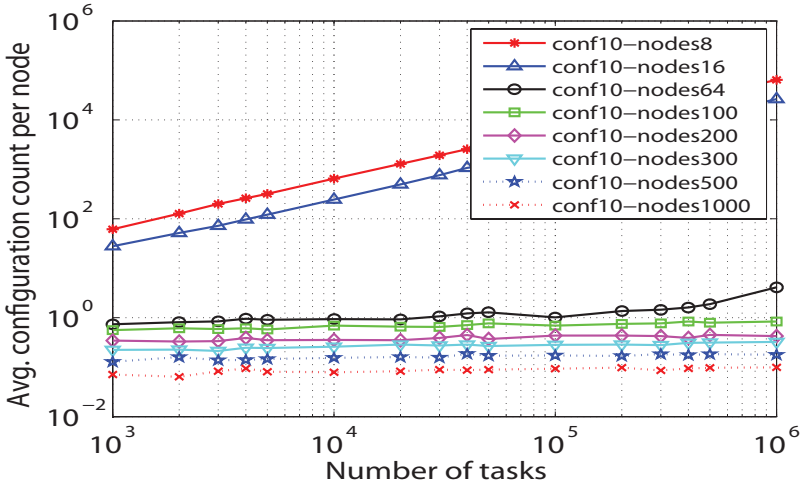
Comparing Figures 5.7 and 5.8, it can be observed that the increase in the task arrival rate reduces the *number of scheduling steps per task* for a given set of tasks, nodes, and configurations. Furthermore, it can be concluded that the more resources (nodes) are available on each *configurations list*, the less scheduling steps are required due to a lower search burden on the *task scheduling module*.

The average reconfiguration count per node: The *average reconfiguration count per node* against given sets of tasks, for fixed set of nodes (8, 16, 64, 100, 200, 300, 500, and 1000 nodes) and 10 configurations, are depicted in Figure 5.9. Figure 5.9 (a) depicts the case when the next task generation is set to [1...50], whereas, Figure 5.9 (b) gives the same results for next task generation interval of [100...1000].

Figure 5.9 (a) depicts that for smaller number of nodes (8, 16, 64, and 100 nodes), the arriving tasks may or may not be accommodated due to insufficient number of nodes. We observed that, due to very fast arrivals of tasks [1...50], and less number of nodes, many tasks are discarded. Consequently, not many (< 1 on average) reconfigurations are performed. When the nodes are sufficient (200, 300, 500, and 1000), the system behaves in a reasonable and expected manner. For relatively less nodes (200 nodes), there is a chance that reconfiguration of certain nodes may be needed due to lesser nodes available per configuration. As a result, average no. of configurations per node is a bit higher (around 2-100 configurations per node) for different sets of tasks. When the number of nodes are more, less configurations per node may be



(a) configurations=10 with task generation interval [1...50].



(b) configurations=10 with task generation interval [100...1000].

Figure 5.9: The *average reconfiguration count per node* for fixed sets of nodes (8, 16, 64, 100, 200, 300, 500, 1000), and 10 configurations. (a) Task arrival rate in interval [1...50] (b) Task arrival rate in interval [100...1000].

required which is indicated by Figure 5.9 (a). However, it should be noticed that for a very large number of nodes (e.g., 1000 nodes), some of the nodes may not be needed at all, thus further decreasing the *average reconfiguration count per node*.

When the arrival rate interval of incoming tasks is relaxed to [100...1000], we can observe constant curves (as depicted in Figure 5.9 (b)) depicting that the *average reconfiguration count per node* is higher for less number of nodes and vice versa. For instance, it is expected that the system will require more number of configurations for less nodes, which is obvious when from the curve for nodes= 8, in Figure 5.9 (b). Similarly, least number of reconfigurations per node are required for the case when nodes = 1000.

From the above mentioned sets of experiments, it is noticed that the system behavior highly depends on the number of resources (nodes) and configurations. Key system metrics such as the *average scheduling steps per task*, the *average waiting time per task*, and the *average reconfiguration count per node* behave in an expected manner for given sets of nodes, configurations and tasks. Similarly, it is observed that the task generation interval greatly affects these key metrics.

5.8 Summary and Conclusion

In this chapter, we presented the design of DReAMSim simulation framework for application task distribution among different nodes of a reconfigurable computing system. We presented and discussed results obtained from our framework which are based on the *average waiting time per task*, the *average scheduling steps required per task*, and the *average reconfiguration count per node*. The framework can be used to investigate the desired system scenario(s) for a given set of parameters, such as tasks, task arrival distributions, nodes, configurations, area ranges, and task required times, etc. The results in this chapter are based on one configuration per node at a given time. In the next chapter, we discuss a scenario of adding multiple configurations per node and compare the results produced by the two scenarios. In case of multiple configurations per node, it is expected that the tasks will have less waiting time on average, and less area per task will be wasted.

Note.

The content of this chapter is based on the following papers:

M. F. Nadeem, I. Ashraf, S. A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements**. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2012)*, Shanghai, China, May 2012.

M. F. Nadeem, S. A. Ostadzadeh, M. Nadeem, S. Wong, and K.L.M. Bertels. **A Simulation Framework for Reconfigurable Processors in Large-scale Distributed Systems**. In *the Proceedings of the 42nd International Conference on Parallel Processing workshops (ICPPW 2011)*, Taiwan, September 2011.

M. F. Nadeem, S. A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **Task Scheduling Strategies for Dynamic Reconfigurable Processors in Distributed Systems**. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, Istanbul, Turkey, July 2011.

6

Scheduling Methodologies utilizing Partial Reconfigurable Nodes

In the previous chapter, a scheduling methodology is discussed that supports task scheduling in a distributed system containing nodes that allowed only one configuration per node at a given time. In this chapter, we present simulation results that are based on *partial* reconfiguration scheduling methodology that supports multiple configurations per node at a given time. We discuss the corresponding scheduling process in detail here. Subsequently, we compare the simulation results in case of *full* and *partial* reconfiguration scenarios.

6.1 Introduction

In [51], we presented an extension of DReAMSim framework to integrate *partial* reconfigurable functionality to the nodes. In this case, only a portion or *region* of a node can be reconfigured at run-time, whereas, the other *regions* are already operative and are executing some task(s). Consequently, the scheduling methodology should also be modified from that of the *full* reconfiguration scheme, where the whole node must be reconfigured instead of only a select *region*. On the other hand, in *partial* reconfiguration scheme, the scheduler should contain the information regarding all the *regions* on each node. If a new task requires the scheduler to reconfigure a new configuration on a node, the scheduler can decide to place it on a node *region*. It first checks the area required by the configuration (`ReqArea`), and then searches for a suitable node *region* that contains an `AvailableArea` which is equal to or more than the `ReqArea` of the configuration.

Similarly, since each node contains multiple configurations, it can be part

of multiple *idle* or *busy* lists of node. The *resource information system* goes through additional burden to keep the updated information regarding all the currently running tasks and their corresponding configurations on all the nodes. Moreover, it has to maintain the dynamic behavior of the *idle* and the *busy* node lists.

Remainder of the chapter is composed of the following sections. In Section 6.2, we describe the scheduling process that supports *partial* reconfigurable nodes. The process is divided into 4 different parts: allocation, configuration, *partial* configuration, and *partial* re-configuration. Section 6.3 provides the simulation environment and results, which are based on the comparison between the *full* and the *partial* reconfiguration scenarios for a given set of parameters. In Section 6.4, we discuss the summary and the conclusions of the chapter.

6.2 Scheduling Process for Partial Reconfigurable Nodes

We tested our framework by implementing a simple dynamic scheduling algorithm, which takes into account the *partial* reconfigurability of the nodes. It was implemented in the scheduler part of the framework. The algorithmic process is mainly divided into four different parts, as depicted in Figure 6.1. Each incoming task is allocated to a particular node by using one of these algorithmic parts namely, *allocation*, *configuration*, *partial configuration*, *partial re-configuration*. Initially, the scheduler decides whether the *exact-match* configuration (or C_{pref}) of the *task* is available in the *configurations list*. If the C_{pref} of the *task* is not available, then the algorithm searches for a *closest-match* configuration (or $C_{ClosestMatch}$) of the *task* in the *configurations list*. However, if $C_{ClosestMatch}$ is also not available, the *task* is discarded. We explain each part of the algorithmic process in the following:

Allocation: If the C_{pref} (or $C_{ClosestMatch}$) is available in the *configurations list*, then the task is directly *allocated* to one of the *idle* nodes already configured with the C_{pref} (or $C_{ClosestMatch}$). The algorithm chooses the *best-match* among all the available idle nodes. The criteria for the *best-match* is the node which possesses the minimum `AvailableArea` among all those nodes which are configured with the C_{pref} (or $C_{ClosestMatch}$), so that the nodes with larger `AvailableArea` are utilized for later *re-configurations*.

Configuration: If the direct *allocation* is not possible due to the absence of

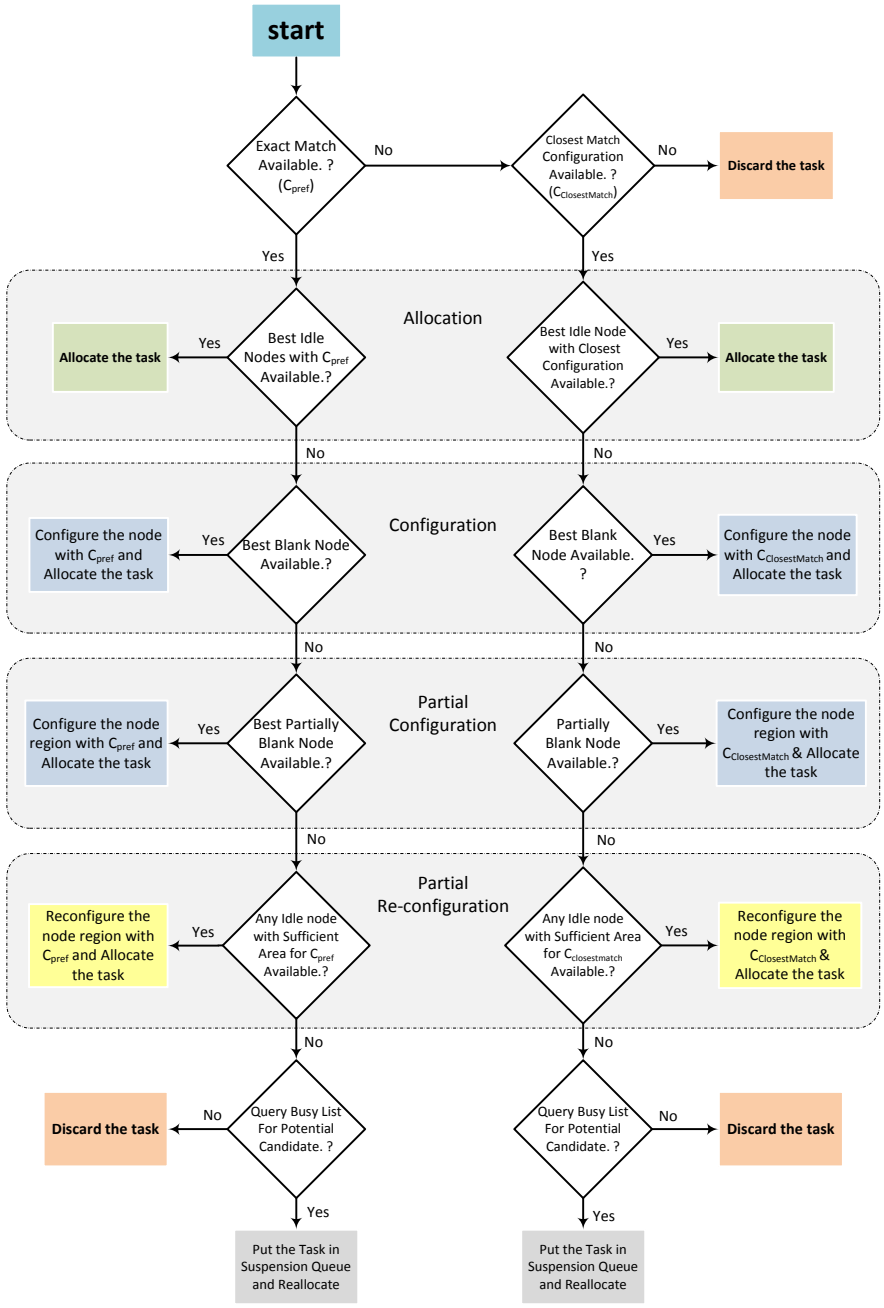


Figure 6.1: The scheduling process with support for *partial* reconfigurability of nodes. There are 4 different phases in the process: (a) *Allocation* (b) *Configuration* (c) *Partial Configuration* and (d) *Partial Re-configuration*.

Algorithm 6.1 The findBestPartiallyBlankNode algorithm

```
Require: task to be scheduled
  bestMatchIndex ← −1
  nodeArea ← 0
  taskArea ← 0
  tempArea ← 0
  minDiff ← MAX_NODE_AREA
  nConfigEntries ← 0
  for counter = 0 → TotalNodes do
    nodeArea ← NodeList[counter].AvailableArea
    taskArea ← task.RequiredArea
    nConfigEntries ← NodeList[counter].ConfigTaskEntries
    tempArea ← nodeArea − taskArea
    if tempArea > 0 &&
    minDiff > tempArea &&
    nConfigEntries < MAX_NODE_CONFIGS then
      bestMatchIndex ← counter
      minDiff ← tempArea
    end if
    SearchLength ← SearchLength + 1
    TotalSimWorkLoad ← TotalSimWorkLoad + 1
  end for
  if bestMatchIndex == −1 then
    return NULL
  else
    return NodeList[bestMatchIndex]
  end if
```

idle node(s), then one of the blank nodes is *configured* with the C_{pref} (or $C_{ClosestMatch}$) and the *task* is allocated. A *blank node* is defined as a node with no current configuration.

Partial configuration: If *allocation* or *configuration* can not be performed, the scheduler searches for a node which contains a reconfigurable region with sufficient area to configure the C_{pref} (or $C_{ClosestMatch}$) of the *task*. In this case, the scheduler chooses a node with minimum sufficient region and configures it with the C_{pref} (or $C_{ClosestMatch}$) and allocates the *task*. This phase is performed the *findBestPartiallyBlankNode()* method in the DReAM-Sim scheduler. This method is presented in Algorithm 6.1. Input to this

Algorithm 6.2 The FindAnyIdleNode algorithm

```
Require: task to be scheduled
  nodeArea ← 0
  Entries ← NULL
  for all node ∈ NodeList do
    accumIdleArea ← node.AvailableArea
    for all entry ∈ node.ConfigTaskList do
      SearchLength ← SearchLength + 1
      TotalSimWorkLoad ← TotalSimWorkLoad + 1
      if entry is idle then
        configArea ← entry.config.RequiredArea
        accumIdleArea ← accumIdleArea + configArea
        Entries ← entry
        if accumIdleArea ≥ task.ReqArea then
          return node
        end if
      end if
    end for
  end for
  return NULL
```

method is the *task* to be scheduled and output is the *node* required for the scheduling of this *task*. As shown in Algorithm 6.1, it searches a node that contains sufficient `AvailableArea` to accommodate this *task*. There can be multiple nodes that satisfy this requirement. However, it chooses the one which has has the minimum remaining area, after the scheduling the *task*. The reason for choosing minimum area criteria is that it will result in utilizing the nodes with lesser remaining area. As a consequence, the left-over nodes can be utilized by the upcoming future tasks which may have big area requirements. A `NULL` is returned, if no nodes with sufficient area are available. In that case, the scheduler performs *partial re-configuration,* as explained below.

Partial re-configuration: In this case, the scheduler explores any idle nodes which are currently configured with any configuration, other than the C_{pref} (or $C_{ClosestMatch}$) of the *task*. This phase is performed by the *FindAnyIdleNode()* method in the scheduler, which is presented by Algorithm 6.2. It takes the *task* as an input and returns a particular node which can be *re-configured* for the scheduling of the task. The method explores the *entries* of the idle

Simulation parameter	Value
Total nodes	100, 200
Total configurations	50
Total tasks generated	1000...100000
Next task generation interval	[1...50]
Configurations *ReqArea* range	[200...2000]
Node *TotalArea* range	[1000...4000]
Task $t_{required}$ range	[100...100,000]
t_{config} range	[10...20]
$C_{ClosestMatch}$ percentage	15%
Reconfiguration method	*with/without partial reconfiguration*

Table 6.1: Various simulation parameters and their values. The reconfiguration methods used are *with/without partial* reconfiguration for two sets of experiments.

configurations which can be removed to reconfigure the node with the C_{pref} (or $C_{ClosestMatch}$) required by *task*. In addition, the method outputs a list of entries which can be utilized for reconfiguration of the node. It returns *NULL* in case no node with sufficient area, can be found for the *re-configuration*.

If the scheduler is unable to *allocate* the *task* after going through all four phases, then it explores the list of all busy nodes to search at least one currently busy node with sufficient *TotalArea* to configure the C_{pref} (or $C_{ClosestMatch}$). If one such node is found, the *task* is put in a suspension queue to later *re-allocate* it to that node to become idle. Otherwise, if no such node is found, the *task* is discarded.

6.3 Simulation Environment and Results

We performed a number of experiments based on the scheduling algorithm implemented in the scheduler part of DReAMSim. We used several simulation parameters to compute various performance metrics. The experiments were conducted on 64-bit Intel Core 2 Duo CPU E8400 machine running at 3.00GHz. It is installed with openSUSE 11.3 with the Linux kernel 2.6.34. The implementation of DReAMSim has been described in C++ and compiled using gcc v.4.5.0 for the above-mentioned target processor.

Simulation parameters: Table 6.1 presents various simulation parameters and their values, used in our experiments. Total number of nodes and con-

figurations are set accordingly for each experiment. The task arrival interval is set between (1—50) *timeticks* with uniform distribution. The total number of tasks in each experiment varies between (1000—100000) and their $t_{required}$ changes between (100—100000) *timeticks* randomly. Furthermore, the `TotalArea` of each node ranges between 1000 to 4000 area units (e.g., area slices). Similarly, all the *configurations* require a reconfiguration area (`ReqArea`) which is set between 200 to 2000 area units. For 15% of the total tasks, we assign a preferred configuration (C_{pref}) that can not be found in *configurations list*. Therefore, these tasks are assigned to the nodes with $C_{ClosestMatch}$ by the scheduler, dynamically.

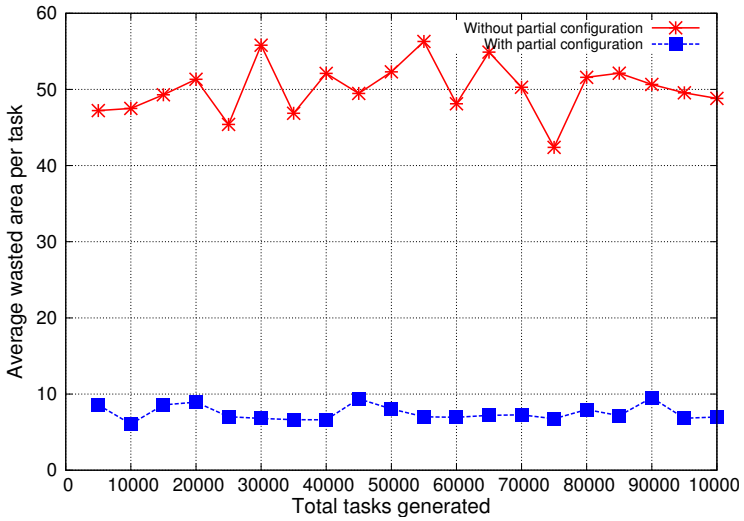
We conducted experiments which are based on two different scenarios. In the first scenario—namely, *without partial configuration*—the nodes can be configured for only one *configuration* at one time. As a result, each node can execute only one task at one time. In the second scenario—namely, *with partial configuration*—each node is able to accommodate as many configurations as possible, depending upon its `AvailableArea`. We extensively tested the functionality of the framework by setting the area requirements of configurations required by all the tasks in such a way that their maximum range is less than or equal to the minimum possible areas on any node. We performed several simulations with various changes in the parameters, and found that no tasks were discarded. Every experiment was conducted several times, and the presented results exhibit the average quantitative values in each cases.

6.3.1 Results Discussion

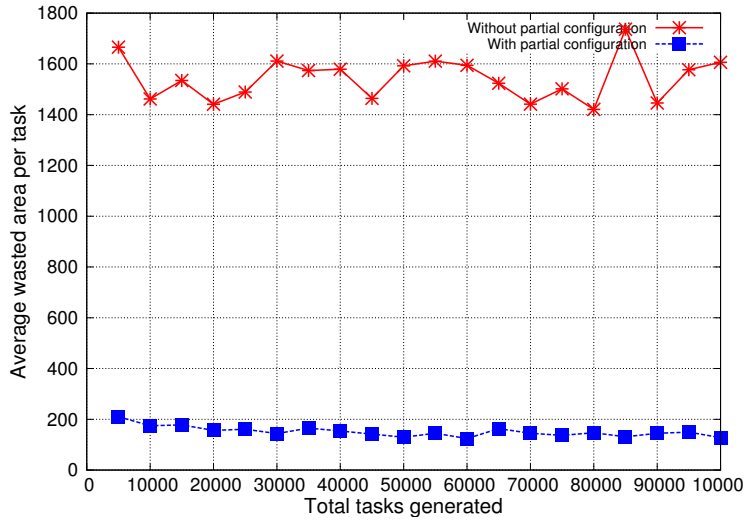
Simulation results for both scenarios mentioned above, are presented and discussed for some key performance metrics such as, *average waiting time per task*, the *average scheduling steps required per task*, and *average reconfiguration count per node*.

The average wasted area per task: Figures 6.2a and 6.2b depict the *average wasted area per task* results against a set of tasks varying between (1000—100000) for 100 and 200 nodes, respectively. All the other parameters are set according to the values in Table 6.1.

First, it can be noticed from both figures that the *average wasted area per task* is less for the scenario *with partial reconfiguration*. This is due to a possibility of adding more tasks on an already-operative node, if it contains sufficient `AvailableArea` to accommodate the incoming task. In case of the scenario *without partial reconfiguration*, only one task can be executed at one time. As



(a) Total nodes=100, total configurations=50. The unit on *y-axis* is *area units*, which quantify the number of *slices* on a reconfigurable node.



(b) Total nodes=200, total configurations=50. The unit on *y-axis* is *area units*, which quantify the number of *slices* on a reconfigurable node.

Figure 6.2: *Average wasted area per task* results for (a) 100 nodes and (b) 200 nodes. In general, lesser reconfigurable area is wasted in the case of *'with partial reconfiguration'* due to more possibilities to accommodate an incoming task. Secondly, more *accumulated* area is wasted in case of 200 nodes.

a result, when the node is reconfigured with the C_{pref} (or $C_{ClosestMatch}$) of the task, the remaining area is wasted and cannot be utilized for another incoming task until the node finishes executing the current task. Consequently, more accumulated area is wasted.

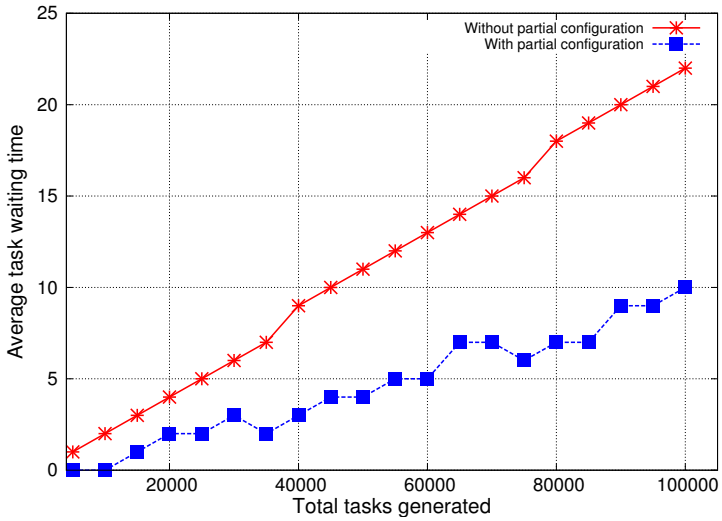
Secondly, the quantitative values of *average wasted area per task* for 100 nodes are far less (10—50 area units) as compared to 200 nodes (200—1600 area units). The reason is that the scheduler has a choice of more number of nodes (200 nodes) for each incoming task. As a result, the total accumulated wasted area is more.

For a fixed set of parameters and 100 nodes, it is expected that the waiting time of each task will be high as compared to 200 nodes. Similarly, it is expected that fewer number of nodes (100 nodes) will be reconfigured more. These results are explained in the following.

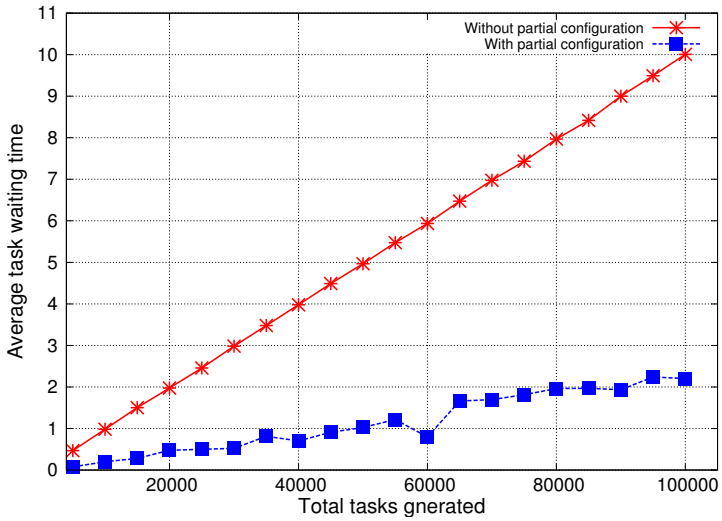
The average waiting time per task: Figures 6.3a and 6.3b depict the *average waiting time per task* results for 100 and 200 nodes, respectively. In the scenario *with partial reconfiguration*, the scheduler can immediately send a task to a particular node if it has sufficient `AvailableArea` to configure its C_{pref} (or $C_{ClosestMatch}$) to one of the *regions* of the node. In this way, it has more options available. Therefore, each incoming task requires less waiting time before it can be scheduled to a suitable node. In a scenario *without partial reconfiguration*, the scheduler has no options to schedule multiple tasks on a single node; therefore, if it cannot find a suitable *idle* node for an incoming task, it puts it into the suspension list until it finds the required node. Thus, the *average task waiting times* are much higher.

In case of 100 nodes, as depicted in Figure 6.3a, the quantitative value of the *average waiting time per task* is very high as compared to 200 nodes (Figure 6.3b). For instance, in case of *'with partial reconfiguration*', this value is 2 *million timeticks* for 100000 tasks and 200 nodes, compared to 10 *million timeticks* for 100 nodes. In general, this value is very high due to a fast arrival rate ((1—50) *timeticks*) of the tasks.

The average reconfiguration count per node: Figures 6.4a and 6.4b depict that, *without partial reconfiguration*, a typical node is reconfigured less times (on average) due to less options available for the scheduler to assign a task to certain node. In this scenario, the scheduler has to wait to assign the task, if all the nodes are *busy*. Therefore the *average waiting time per task* increases considerably, which is clear from the corresponding results above. However, in the scenario of *'with partial reconfiguration*', there are more node *regions* available and the scheduler has more options to assign the task to a

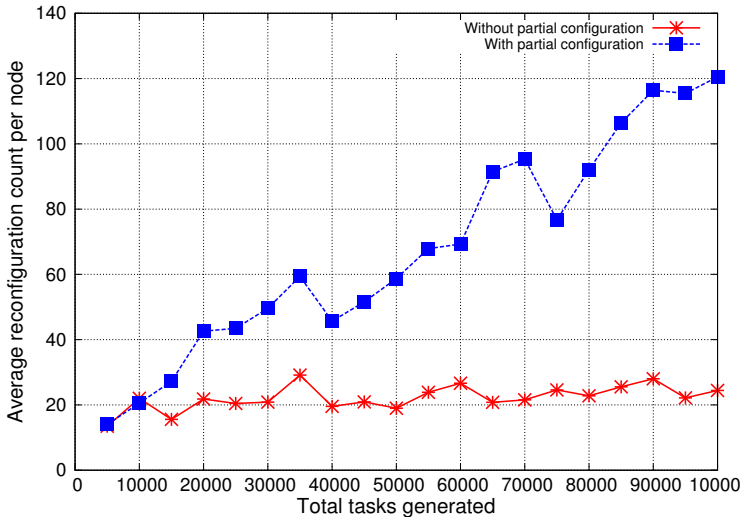


(a) Total nodes=100, total configurations=50. The unit on *y-axis* is in million *timeticks*, quantifying the progression of time on the host system.

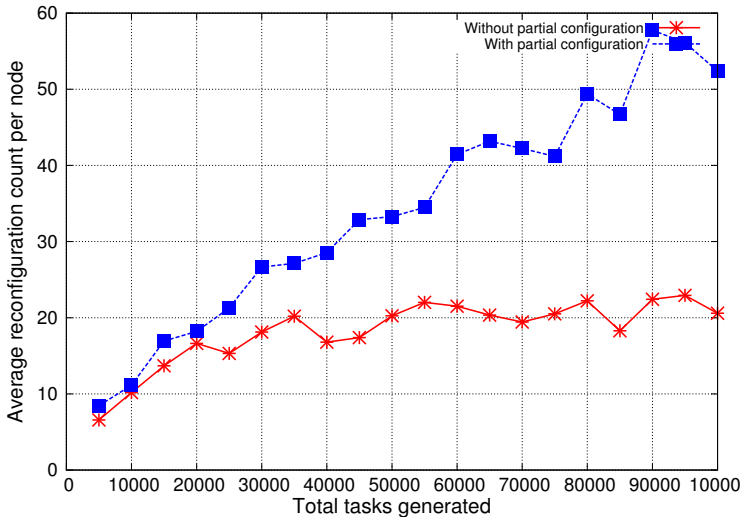


(b) Total nodes=200, total configurations=50. The unit on *y-axis* is in million *timeticks*, quantifying the progression of time on the host system.

Figure 6.3: *Average waiting time per task* results for (a) 100 nodes and (b) 200 nodes. In case of *'with partial reconfiguration'*, an incoming task suffers less wait on average, as the scheduler can assign it to a node *region*, unlike in case of *'without partial reconfiguration'*.



(a) Total nodes=100, total configurations=50. The unit on *y-axis* quantifies the average number of reconfigurations of a node.



(b) Total nodes=200, total configurations=50. The unit on *y-axis* quantifies the average number of reconfigurations of a node.

Figure 6.4: *Average reconfiguration count per node* results for (a) 100 nodes and (b) 200 nodes. In case of *'with partial reconfiguration'*, the reconfiguration count is high due to more options to assign a task to a node *region*, rather than waiting for the whole node as in case of *'without partial reconfiguration'*.

particular node *region*. Hence, the *average reconfiguration count per node* is high as compared to the scenario when only one *configuration* (or one task) is allowed at one time. In general, this increase in the *reconfiguration count per node* involves many trade-off in reducing the *average waiting time per task* and the overall *tasks completion time*, which are discussed in the next subsection.

Furthermore, it can be noticed that as the number of tasks increase, the average quantitative values of the *reconfiguration count* also increase (see Figures 6.4a and 6.4b). It is because, the number of nodes is limited (100 and 200 nodes), while the number of tasks is increasing (1000—100000 tasks). Therefore, a typical node will go through more number of reconfigurations for a higher number of tasks.

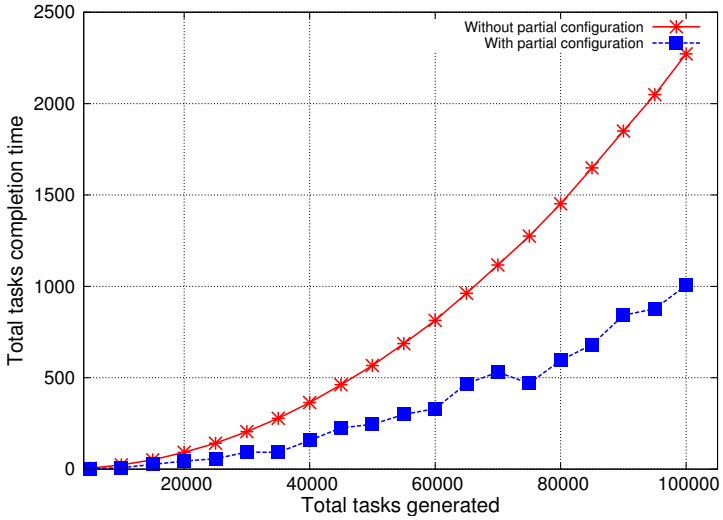
Finally, in case of 100 nodes, as depicted in Figure 6.4a, the *reconfiguration count* is relatively higher than that of 200 nodes. It is because the scheduler has less options to schedule each incoming task. It explores any *idle* nodes which are currently configured with any configuration, other than the C_{pref} (or $C_{ClosestMatch}$) of the task. As a result, it reconfigures those *idle* nodes with C_{pref} (or $C_{ClosestMatch}$) of the current task, and schedules the task. Hence, the quantitative value of the *average reconfiguration count per node* is high in case of 100 nodes.

The total tasks completion time: Figures 6.5a and 6.5b depict the *total completion time* results for all tasks for the two scenarios in case of 100 and 200 nodes, respectively. From Equation 5.9, it is clear that, the completion time largely depends on the waiting time of the tasks.

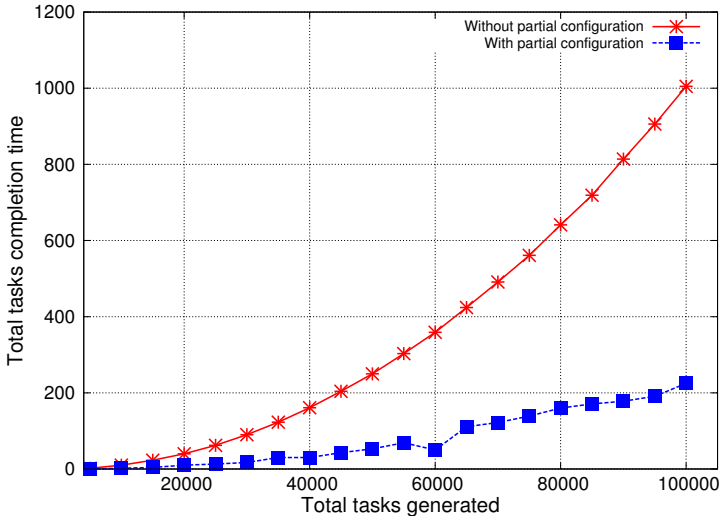
As shown earlier that the *average task waiting time* is high in the scenario *without partial reconfiguration*, which results in an increase in the *total completion time* of the tasks. In the case *with partial reconfiguration*, the scheduler has more options to assign the tasks to a node *region*. Although the *reconfiguration count per node* increases, but the tasks do not have to wait in the suspension queue for their scheduling. Their average waiting time decreases; hence a considerable reduction in the *total completion time*.

It is also evident from Figures 6.5a and 6.5b, that the *total tasks completion time* is very high in case of 100 nodes, as compared to 200 nodes. For instance, the quantitative value (expressed in *billion timeticks*) is 1000 as compared to 200, for the case *with partial reconfiguration* in two figures. It is due to larger task waiting times in case of 100 nodes.

The total scheduler workload: Figures 6.6a and 6.6b depict the *total scheduler workload* results, expressed in terms of search steps in billions, for 100

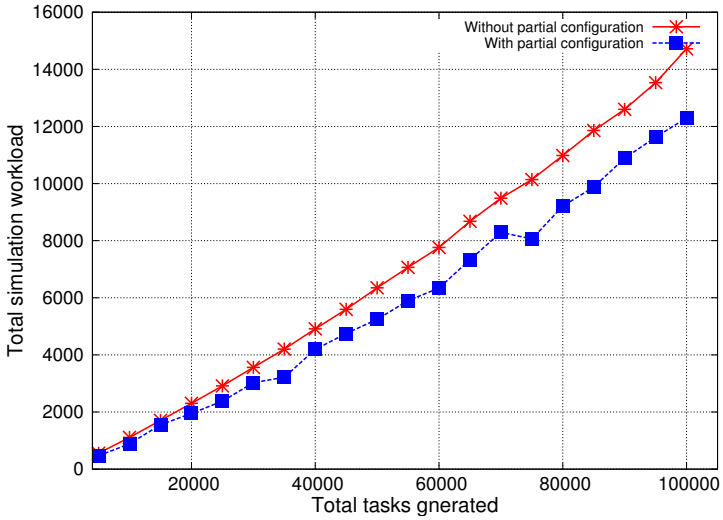


(a) Total nodes=100, total configurations=50. The unit on *y-axis* is expressed in billion *timeticks*.

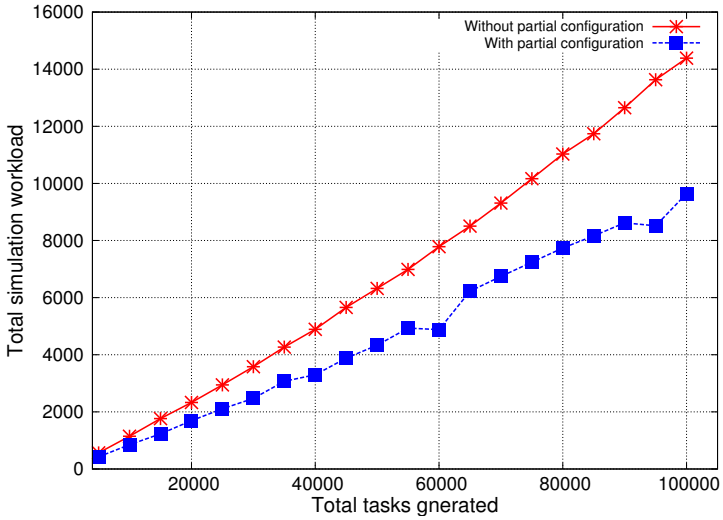


(b) Total nodes=200, total configurations=50. The unit on *y-axis* is expressed in billion *timeticks*.

Figure 6.5: *Total task completion time* results for (a) 100 nodes and (b) 200 nodes. The overall task completion time is less in case of *'with partial reconfiguration'* due to relatively short task waiting times.



(a) Total nodes=100, total configurations=50. The unit on *y-axis* is the count of *search steps* (in billions) taken by the scheduler during a simulation.



(b) Total nodes=200, total configurations=50. The unit on *y-axis* is the count of *search steps* (in billions) taken by the scheduler during a simulation.

Figure 6.6: *Total scheduler workload* results for (a) 100 nodes and (b) 200 nodes. In case of *'with partial reconfiguration'*, a lesser number of scheduling steps are required to assign a task to a certain node.

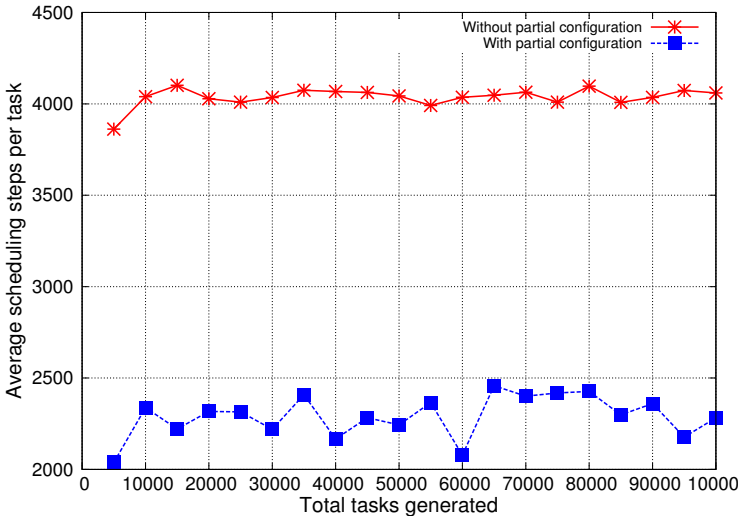


Figure 6.7: *Average scheduling steps per task* results for 200 nodes and total configurations=50. The unit on *y-axis* is the count of average number scheduling steps taken by the scheduler to accommodate a task. In case of *'with partial reconfiguration'*, a lesser number of scheduling steps are required to assign a task to a certain node.

and 200 nodes, respectively. It can be noticed that the scheduler endures more workload in the scenario *without partial reconfiguration*. This is because, the possibilities to schedule a task are limited and more housekeeping is required. The workload metric also includes the number of times the scheduler checks the suspended tasks list. In case *without partial reconfiguration*, more tasks are added to this list; thus resulting in an increase in the workload. On the other hand, in the scenario *with partial reconfiguration*, although less tasks are added to the suspension list, the scheduler has to maintain higher number of *idle* and *busy* lists due to node *regions* being reconfigured. For this reason, the two graphs are very close to each other.

If we compare Figures 6.6a and 6.6b, it is apparent that the total scheduler workload is less in case of 200 nodes. In this case, although the scheduler has to maintain higher number of lists, it has more number of nodes to accommodate a tasks.

The average scheduling steps per task: The *average scheduling steps per task* result comparison between the two scenarios for 200 nodes, are depicted in Figure 6.7. In the first case *with partial reconfiguration*, the scheduler can

even search for a node *region* to assign a task, which reduces the scheduling effort to accommodate a task. This results in less scheduling steps as compared to the case *without partial reconfiguration*.

Results summary

We presented and discussed several simulation results for two scenarios of *with/without partial reconfiguration,* considering a given set of parameters (see Table 6.1), and varying number of nodes from 100 to 200. Here, we give a brief summary of these results.

With/without *partial* reconfiguration scenarios

We examined these two scenarios for many performance metrics of the simulator. In a nutshell, it can be concluded that the system utilizes lesser area, scheduler workload, and scheduling steps, in case of *with partial reconfiguration*. Since each node can accommodate multiple tasks, it goes through a higher number of reconfigurations, resulting in lesser average waiting times, total tasks completion times, and the average wasted reconfigurable area. Furthermore, it can be ascertained that the reconfiguration methods thoroughly affect the behavior of the system, and all the performance metrics follow an expected trend for both the scenarios.

100/200 number of nodes

When we increased the number of nodes from 100 to 200, the curves representing the metrics reciprocated accordingly. The *average wasted area per node* increased, whereas, the corresponding curves for the task waiting times, decreased. Similarly, the reconfiguration count decreased because of the availability of more nodes for each task, resulting in lesser node reconfigurations. Moreover, the corresponding values for the *total tasks completion time* and the *total scheduler workload* are reduced in case of 200 nodes.

6.4 Summary and Conclusion

In this chapter, we presented the design of a generic scheduling process to investigate the performance of *partial* reconfigurable processors in distributed systems. We implemented the scheduling methodology in our simulation framework and generated several simulation results based on two different scenarios. In the first scenario, the nodes allowed *partial* reconfiguration, whereas, in the second scenario, the nodes allowed only *full* reconfiguration.

The results suggested that the *average wasted area per task* in case of *partial* reconfiguration scenario, is less as compared to full configuration. Hence, the node utilization can be improved by adding more than one reconfigurations on a node. Furthermore, the *average waiting time per task* for the *partial* reconfiguration scenario are less as compared to the *full* reconfiguration. As expected, the *partial* reconfiguration case has a higher *reconfiguration count per node*.

Note.

The content of this chapter is based on the following paper:

M. F. Nadeem, I. Ashraf, S. A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements**. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2012)*, Shanghai, China, May 2012.

7

Conclusions and Future Work

This dissertation highlighted the necessary background information, motivation, and research methodology to investigate the task scheduling policies in a distributed system containing reconfigurable nodes. For this purpose, we proposed a generic virtualization framework, and the design of resource information system. Next, we presented the details of DReAMSim framework, and tested two variants of a scheduling process that supports task scheduling in distributed reconfigurable systems. In Section 7.1 of this chapter, we summarize the work presented in the earlier chapters of this dissertation. Next, we present the main contributions of this thesis in Section 7.2. Finally, in Section 7.3, we outline some open issues and future directions based on this research work.

7.1 Summary and Conclusions

This dissertation is mainly focused around the development of a simulation framework to evaluate the performance of task scheduling algorithms in distributed systems utilizing reconfigurable processors. **In Chapter** 1, the significance of high-performance distributed computing systems is discussed, along with the feasibility of merging reconfigurable processors in these systems. In this respect, we analyzed the research challenges and provided a problem overview. We discussed that the ongoing research in reconfigurable computing technology can open up new possibilities to utilize it in high-performance computing. Furthermore, there are currently very limited number of simulation tools to analyze the various aspects of such systems. In the 1st chapter, we briefly gave an overview of our proposed simulation framework and discussed our research methodology. Finally, we conclude the chapter with a summary of dissertation organization.

In Chapter 2, we presented a generic background of high-performance distributed computing systems. As an example, we outlined an important enabling environment, known as the *volunteer computing*, where many voluntary Internet users can collaborate together to build up a distributed computing system, that can solve a large-scale scientific problem. We highlighted its computing power and various real-world systems utilizing volunteer computing environment. We also specified major efforts made in developing the middleware technologies, that can provide the necessary resource coupling facilities to develop a real testbeds. Subsequently, we briefly characterized the recent success of the reconfigurable computing technology. We also discussed the benefits of *partial* reconfigurable systems. Furthermore, we outlined several projects utilizing reconfigurable computing in distributed system domain. They include customized setups, reconfigurable cluster systems, grid computing, and high-performance multi-FPGA systems. Then, we briefly discussed scheduling in distributed systems by enlisting important schedulers in real-world computing testbeds. Next, we gave a detailed overview of numerous simulation tools in this field. We presented a comparison between these tools in terms of their main features, resource models, task representation, application models, cost, flexibility, documentation, and development language. Finally, we pointed out essential parameters required to develop a tool that can assist in the evaluation of scheduling algorithms in distributed systems. The main conclusion of the chapter is that new simulation tools are required to incorporate these parameters.

In Chapter 3, we provided a generic virtualization framework for **R**econfigurable **P**rocessing **E**lements (RPEs) in distributed systems. We outlined the concept and necessary background information, and then enlisted numerous use-case scenarios of applications that can utilize RPEs. We described how can various types of application take benefit from RPEs in a distributed system. Subsequently, we presented the virtualization framework that encompasses a generic node model which can accommodate both RPEs and GPPs. Moreover, it offers a model for an application task that can make use of such a node. Finally, we presented a case-study from a real-world application, and described all possible mappings of its task to nodes in a system.

In Chapter 4, we illustrated—with the help of a detailed motivational example—the Resource Information Services (RIS) required to accommodate reconfigurable processors in distributed systems. We introduced the basic concepts of RIS, and highlighted some key previous developments in this field. Then, we formulated a formal system model and RIS data structures that provide the basis to our proposed simulation framework, which is detailed in

Chapter 5 and Chapter 6. Subsequently, we provided a detailed example that illustrated the basic mechanism of RIS used in our simulation framework. The example is based on two different scenarios of reconfigurable nodes in the system. In the first case, the nodes are allowed to have one configuration, and in the second case, each node can accommodate multiple configurations. From this chapter, we concluded that novel data structures are required in order to maintain the information related to reconfigurable nodes in a distributed system.

In Chapter 5, we presented the design and implementation details of our proposed simulation framework. We discussed various subsystems in the framework, and defined several terminologies used throughout the chapter. Next, we outlined the detailed description of the design and implementation of the RIS data structures, keeping in view the motivational example in Chapter 4. Subsequently, we presented the UML model of the framework, and highlighted all the important classes and methods that constitute the design. Then, we enlisted several performance metrics that can be generated by the framework. It is described how each metric is computed during the simulation process. We implemented a simple task scheduling algorithm that takes into account the reconfigurable nodes, to test the functionality of the framework. For a given set of parameters, we generated many simulation results and discussed them in detail. We used nodes without partial reconfiguration. Finally, we provided various conclusions based on the simulation results.

In Chapter 6, we presented some more results based on a variant of scheduling process given in Chapter 5. In this case, the nodes can support *partial* reconfiguration. For this purpose, the scheduling methodology should accommodate multiple configurations at a given time. Consequently, we implemented a scheduling algorithm that supports *partial* reconfigurable nodes. We highlighted the algorithmic modifications required in this case. Based on the algorithm, we generated simulation results for a given set of parameters, and discussed them in detail. Mainly, we compared two different sets of results for the *full* and the *partial* reconfiguration of nodes.

The experimental results demonstrate that the *average wasted area per task* in case of utilizing partial reconfigurable nodes is lesser as compared to *full* reconfigurable nodes. Similarly, it is concluded that for a given set of parameters, the system takes less scheduling time and less waiting time. Therefore, the nodes can be utilized in a better way, by adding more than one configurations per node. The framework can be used to simulate many scheduling

policies for any set of parameters, for instance nodes, tasks, configurations, task arrival rates, task arrival distributions, area bounds, etc.

7.2 Main Contributions

This dissertation is primarily focused on the design and development of the DReAMSim framework. It addressed the open research problems mentioned in the introduction chapter, and provided the following specific contributions.

Contribution 1—A thorough overview of existing simulation tools, real computing test-beds, and scheduling strategies in both distributed and embedded domains.

We provided a detailed overview of the current and future distributed computing systems that contain RPEs. Moreover, we discussed their advantages and limitations. We also gave a brief summary of the existing simulation tools and the real test-beds in this domain. The survey served as a motivation behind the research work presented in this dissertation.

Contribution 2—Design and development of a simulation framework in order to simulate the task scheduling strategies in a distributed computing system containing RPEs.

The DReAMSim framework is generic and simple, and it allows to model RPEs, application tasks, and processor configurations utilizing different parameters, which include reconfigurable area, reconfiguration delays, network delays, task arrival distributions, task execution times, and reconfiguration methods etc. It implements a resource information service to maintain the information of all the nodes. Moreover, it allows to test different scheduling policies for task distributions among the nodes. It also employs partial reconfiguration functionality to the nodes, where some *node regions* can be reconfigured at run-time while the other *region(s)* are busy in processing some task(s). As a result, multiple tasks can be executed by each node. The framework is developed in a modular fashion to allow more extensions in the future.

Contribution 3—Proposal of a task scheduling system, that allows various task scheduling strategies to assign tasks to RPEs in a distributed computing system.

We proposed and implemented a task scheduling system in the DReAMSim, which allows to test various task scheduling strategies in a distributed sys-

tem. The strategies also support the scheduling of the tasks on partially reconfigurable nodes.

Contribution 4—Design and implementation of data structures allowing the information maintenance in case of RPEs in a distributed computing system.

We described the resource management issues in a distributed system, in the context of adding RPEs as a computing resource. In this respect, we provided a system model formulation. Furthermore, we described a simple methodology to maintain the information regarding the resource nodes, and it is illustrated with the help of a detailed motivational example. We gave the design of a *Resource Information System* (RIS), that implements the necessary data structures required to update the information. These data structures contain the information corresponding to various entities in the system.

Contribution 5—Proposal of virtualization models for RPEs in distributed computing systems.

We presented various use-case scenarios to discuss the utilization of RPEs in a distributed system. Based on these scenarios, we presented a generic virtualization framework for a distributed computing system that supports RPEs, along with the GPPs. Moreover, we presented a typical application task model that can utilize various types of PEs.

7.3 Open Issues and Future Directions

Although the simulation framework proposed in this dissertation provides a basis to evaluate many different scheduling strategies for a distributed system containing reconfigurable processing nodes, there are still many areas where our approach can be further extended. In the following, we enlist a number of open issues and future directions based on the work presented in this dissertation.

- With the emergence of distributed computing paradigm, there is a huge research emphasis on market-based economy in grids [124] [125] [126]. In an economic model, a hypothetical market is developed based on resource trading, where resource producers can sell their computing cycles to the consumers, based on some match-making strategy [127]. Many simulation tools provide an opportunity to test auction-based algorithms for resource trading [45] [83]. In this respect, it is interesting to introduce reconfigurable devices as another novel resource

and the resource trading is performed on the basis of reconfigurable area, rather than cycles. Our simulation framework can be extended to allow the testing of resource trading based on reconfigurable area.

- In this dissertation, we discussed only two variants of a generic scheduling process that allows the task scheduling on reconfigurable processors. However, several other scheduling strategies can be implemented, based on different variants of task arrival rates, task arrival strategies, best-match resources, load-balancing, greedy scheduling etc. Currently, the scheduling assumes the application in the form of "*bag of tasks*", where each task is considered as an independent task, but the simulator can be extended to allow inter-task communication. Similarly, task preemption can be added to the framework.

- In this dissertation, we used arbitrary parameters, and synthetic sets of tasks during our simulations. However, real-world workloads can be used for future work. It will require an extensive application model where tasks require specific hardware for their execution. The research in this field is still at its rudimentary phase, but such workloads and applications will arise, once real testbeds are investigated, in which reconfigurable processors are frequently utilized as computing resource.

- The framework can be extended to develop a **G**raphical **U**ser **I**nterface (GUI) and a visualization system. With such development, a user can input all the parameters using the GUI and can visualize and monitor the results on run-time. Many generic load-balancing policies can be added to the GUI to visualize various aspects of a scheduling algorithm.

- In its current implementation form, the simulation framework assumes a uniform network between the scheduler and nodes. This network is represented in form of communication delays which are spawned once a task and/or bitstream transfers are made. This is an extremely abstract representation of a real-world network. The focus of this dissertation was to provide a basic framework to incorporate reconfigurable processors in a simulation tool. However, in real-world systems, the communication networks are of paramount importance. Therefore, it will be an important future work to extend the framework for modeling a more realistic network in terms of routers, bandwidth, background traffic etc.

Bibliography

[1] Web page. BOINC Statics. Project Stats Info, Last Checked: 23-03-2013. http://boincstats.com/en/stats/projectStatsInfo.

[2] Web page. The Top500 Supercomputing Projects, Last Checked: 23-03-2013. http://www.top500.org/.

[3] Web page. Open Grid Forum. Grid Projects, Last Checked: 29-04-2013. http://www.ogf.org/UnderstandingGrids/grid_projects.php.

[4] R. Kurzweil. The Singularity is Near: When Humans Transcend Biology. *Penguin (Non-Classics)*, 2006.

[5] J.P. Morrison et al. Architecture and Implementation of a Distributed Reconfigurable Metacomputer. In *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing*, pages 153–158, 2003.

[6] D. Koch, M. Koerber, and J. Teich. Searching RC5-Keys with Distributed Reconfigurable Computing. In *Proceedings of the 2006 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 42–48, 2006.

[7] J.P. Walters et al. MPI-HMMER-Boost: Distributed FPGA Acceleration. *Journal of VLSI Signal Processing System*, 48(3):223–238, September 2007.

[8] R. Sass et al. Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–140, 2007.

[9] D. Stainforth et al. Distributed Computing for Public-Interest Climate Modeling Research. *Computing in Science and Engineering*, 4(3):82–89, 2002.

[10] E. Krieger and G. Vriend. Models@Home: Distributed Computing in Bioinformatics Using a Screensaver Based Approach. *Bioinformatics*, 18(2):315–318, 2002.

[11] R. Buyya, K. Branson, J. Giddy, and D. Abramson. The Virtual Laboratory: A Toolset to Enable Distributed Molecular Modelling for Drug

Design on the World-Wide Grid. *Concurrency and Computation: Practice and Experience*, 15(1):1–25, 2003.

[12] D.P. Anderson et al. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[13] V. Agarwal, D.A. Bader, and L. Liu. Financial Modeling on the Cell Broadband Engine. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[14] D.J. Groen. High-Performance N-body Simulation on Computational Grids. *PhD Thesis*, 2010.

[15] B. Schmidt. A Survey of Desktop Grid Applications for e-Science. *International Journal of Web Grid Services*, 3(3):354–368, August 2007.

[16] The BlueGene/L Team, T. Domany et al. An Overview of the BlueGene/L Supercomputer, 2002.

[17] Baxter et al. Maxwell- A 64 FPGA Supercomputer. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 287–294, 2007.

[18] G. Blake, R.G. Dreslinski, T. Mudge. A Survey of Multicore Processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[19] L.F.G. Sarmenta. Volunteer Computing. *PhD Thesis*, 2001.

[20] I. Foster and C. Kesselman. Computational Grids. In *Selected Papers and Invited Talks from the 4th International Conference on Vector and Parallel Processing (VECPAR)*, pages 3–37, 2001.

[21] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[22] Web page. The Cascade. Cray XC30 Supercomputer Series - The Cascade Program, Last Checked: 29-04-2013. `http://www.cray.com/Programs/Cascade.aspx`.

[23] Web page. IBM Intelligent Cluster, Last Checked: 29-04-2013. `http://www-03.ibm.com/systems/x/hardware/largescale/cluster/index.html`.

[24] W.W. Wilcke et al. IBM Intelligent Bricks Project: Petabytes and Beyond. *IBM Journal of Research and Development*, 50(2/3):181–197, March 2006.

[25] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *First IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pages 35–39. IEEE Computer Society, 2001.

[26] F. Allen et al. Blue Gene: A Vision for Protein Science using a Petaflop Supercomputer. *IBM System Journal*, 40(2):310–327, February 2001.

[27] R. Buchty, V. Heuveline, W. Karl, and J.P. Weiss. A Survey on Hardware-Aware and Heterogeneous Computing on Multicore Processors and Accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012.

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

[29] Web page. Xilinx Inc. The Virtex-7 Series FPGAs Documentation, Last Checked: 23-03-2013. `http://www.xilinx.com/support/documentation/7_series.htm`.

[30] Web page. Altera. The Stratix-V FPGAs Family, Last Checked: 23-03-2013. `http://www.altera.com/devices/fpga/stratix_fpgas/stratix_v/stxv_index.jsp`.

[31] Web page. Xilinx Inc. The Virtex-7 Series FPGAs Data Sheets, Last Checked: 23-03-2013. `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.

[32] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design and Test of Computers*, 22(2):114–125, 2005.

[33] K. Bondalapati and V.K. Prasanna. Reconfigurable Computing Systems. In *Proceedings of the IEEE*, volume 90, pages 1201–1217, 2002.

[34] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–134, 2010.

[35] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing. *ACM Transactions in Reconfigurable Technology Systems*, 1(4):1–23, 2009.

[36] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model. *ACM Transactions on Reconfigurable Technology Systems*, 4(4):36:1–36:24, December 2011.

[37] R. Karanam et al. Using FPGA-Based Hybrid Computers for Bioinformatics Applications. *Xcell Journal*, pages 80–83, 2006.

[38] El-Ghazawi et al. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41:69–76, February 2008.

[39] K.H. Tsoi and L. Wayne. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 115–124, 2010.

[40] M. Showerman et al. QP: A Heterogeneous Multi-Accelerator Cluster. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[41] F. Dong and S.G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. *Technical Report No. 2006-504*, 2006.

[42] H. Cao et al. DAGMap: Efficient and Dependable Scheduling of DAG Workflow Job in Grid. *Journal of Supercomputing*, 51:201–223, February 2010.

[43] H. Topcuoglu, S. Hariri, and M.Y. Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW)*, pages 3–17, 1999.

[44] N. Muthuvelu et al. A Dynamic Job Grouping-based Scheduling for Deploying Applications with Fine-grained Tasks on Global Grids. In *Proceedings of the 2005 Australasian Workshop on Grid Computing and e-Research (ACSW Frontiers)*, pages 41–48, 2005.

[45] 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, 2002.

[46] K. Aida et al. Performance Evaluation Model for Scheduling in Global Computing Systems. *International Journal of High Performance Computing Applications*, 14(3):268–279, 2000.

[47] H.J. Song et al. The MicroGrid: A Scientific Tool for Modeling Computational Grids. *Journal of Scientific Programming*, 8(3):127–141, 2000.

[48] H. Casanova. SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 430–437, 2001.

[49] S. Wong and M. Ahmadi. Reconfigurable Architectures in Collaborative Grid Computing: An Approach. In *Proceedings of the 2nd International Conference on Networks for Grid Applications (GridNets)*, 2008.

[50] M.F. Nadeem et al. A Simulation Framework for Reconfigurable Processors in Large-Scale Distributed Systems. In *Proceedings of the 40th International Conference on Parallel Processing Workshops (ICPPW)*, pages 352–360, 2011.

[51] M.F. Nadeem et al. Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2012.

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

[53] M.F. Nadeem, M. Ahmadi, M. Nadeem, and S. Wong. Modeling and Simulation of Reconfigurable Processors in Grid Networks. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 226–231. IEEE, 2010.

[54] R. Baldwin et al. Queueing Network Analysis: Concepts, Terminology, and Methods. *Journal of Systems and Software*, 66(2):99–117, 2003.

[55] D.P. Anderson. BOINC: A System for Public Resource Computing and Storage. In *the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[56] C. Christensen, T. Aina, and D. Stainforth. The Challenge of Volunteer Computing with Lengthy Climate Model Simulations. In *1st International Conference on e-Science and Grid Computing*, pages 8–15. IEEE, 2005.

[57] Web Page. The BioGrid Center Kansai, Last Checked: 29-04-2013. `http://www.biogrid.jp/eng/index.html`.

[58] F. Gagliardi. The EGEE European Grid Infrastructure Project. *High Performance Computing for Computational Science - VECPAR*, 3402:194–203, 2005.

[59] M. Ellert et al. The NorduGrid Project: using Globus Toolkit for Building GRID Infrastructure. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 502(3):407–410, 2003.

[60] Web page. Extreme Science and Engineering Discovery Environment (XSEDE), Last Checked: 29-04-2013. `https://www.xsede.org/`.

[61] R. Buyya and S. Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report. In *1st IEEE International Workshop on Grid Economics and Business Models*.

[62] D. Erwin and D. Snelling. UNICORE: A Grid Computing Environment. *Euro-Par Parallel Processing*, pages 825–834, 2001.

[63] S.J. Chapin et al. The Legion Resource Management System. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 162–178, 1999.

[64] K. Regester, J. Byun, A. Mukherjee, and A. Ravindran. Implementing Bioinformatics Algorithms on Nallatech-configurable Multi-FPGA Systems. *Xcell Journal*, (53):100–103, 2005.

[65] S. Dydel, K. Benedyczak, and P. Bala. Enabling Reconfigurable Hardware Accelerators for the Grid. In *Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, PARELEC '06, pages 145–152, 2006.

[66] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded Processor. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 275–285, 2001.

[67] W. Lie and W. Feng-yan. Dynamic Partial Reconfiguration in FPGAs. In *the 3rd International Symposium on Intelligent Information Technology Application*, volume 2, pages 445–448, 2009.

[68] C. Kao. Benefits of Partial Reconfiguration. *Xcell Journal*, 55:65–67, 2005.

[69] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Performance Bounds of Partial Run-time Reconfiguration in High-performance Reconfigurable Computing. In *Proceedings of the 11th International Workshop on High-performance Reconfigurable Computing Technology and Applications*, pages 11–20, 2007.

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

[71] J.P. Morrison, P.J.O. Dowd, and P.D. Healy. Searching RC5 Keyspaces with Distributed Reconfigurable Hardware. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 269–272, 2003.

[72] R.L. Rivest. The RC5 Encryption Algorithm. *Fast Software Encryption*, 1008:86–96, 1995.

[73] S. Eddy. HMMER: Profile HMMs for Protein Sequence Analysis. *HMMER: Sequence Analysis using Pofile Hidden Markov Models Web Site*, 2003.

[74] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message Passing Interface. *MIT press*, 1, 1999.

[75] P. Rice et al. EMBOSS: the European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, 2000.

[76] S.L. Graham et al. Gprof: A Call Graph Execution Profiler. *SIGPLAN Notices*, 17:120–126, 1982.

[77] L. Moll, A. Heirich, and M. Sh. Sepia: Scalable 3D Compositing using PCI Pamette. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 1999.

[78] R. Sass, K. Underwood, and W.B. Ligon III. Design of the Adaptable Computing Cluster. In *Proceedings of the 4th Annual Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 2001.

[79] K. Datta and R. Sass. Rboot: Software Infrastructure for a Remote FPGA Laboratory. In *the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 343–344, 2007.

[80] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW)*, pages 349–363, 2000.

[81] S. Venugopal, R. Buyya, and L. Winton. A Grid Service Broker for Scheduling Distributed Data-oriented Applications on Global Grids. In *Proceedings of the 2nd Workshop on Middleware for Grid Computing*, pages 75–80, 2004.

[82] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-pacific Region (HPC ASIA)*, pages 283–289. IEEE Computer Society Press, 2000.

[83] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17:2–4, 2005.

[84] H. Casanova and J. Dongarra. NetSolve: A Network-enabled Server for Solving Computational Science Problems. *International Journal of High Performance Computing Applications*, 11(3):212–223, 1997.

[85] B. Chun et al. Planetlab: An Overlay Testbed for Broad-coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[86] W.H. Bell et al. Optorsim: A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications*, 17(4):403–416, 2003.

[87] K. Vanmechelen, W. Depoorter, and J. Broeckhove. A Simulation Framework for Studying Economic Resource Management in Grids. *Computational Science–ICCS*, pages 226–235, 2008.

[88] A. Takefusa. Bricks: A Performance Evaluation System for Scheduling Algorithms on the Grids. In *the JSPS Workshop on Applied Information Technology for Science (JWAITS)*, 2001.

[89] C.L. Dumitrescu and I. Foster. GangSim: A Simulator for Grid Scheduling Studies. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, volume 2, pages 1151–1158, 2005.

[90] T. Issariyakul and E. Hossain. Introduction to Network Simulator NS2. *Springer*, 2011.

[91] A. Varga. The OMNeT++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.

[92] J. Cowie, A. Ogielski, and D. Nicol. The SSFNet Network Simulator. *Software On-line: http://www.ssfnet.org/homePage.html*, 2002.

[93] X. Chang. Network Simulations with OPNET. In *Simulation Conference Proceedings*, volume 1, pages 307–314, 1999.

[94] F. Gagliardi, B. Jones, M. Reale, and S. Burke. European DataGrid Project: Experiences of Deploying a Large scale Testbed for e-Science Applications. *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 255–264, 2002.

[95] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[96] 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)*, pages 43–50, 2008.

[97] M. Ahmadi, A. Shahbahrami, and S. Wong. Collaboration of Reconfigurable Processors in Grid Computing for Multimedia Kernels. pages 5–14, 2010.

[98] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid - Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15:1–24, 2001.

[99] C. Plessl and M. Platzner. Virtualization of Hardware - Introduction and Survey. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 63–69, 2004.

[100] C.H. Huang and P.A. Hsiung. Hardware Resource Virtualization for Dynamically Partially Reconfigurable Systems. *IEEE Embedded Systems Letters*, 1(1):19–23, 2009.

[101] Y. Ha et al. A Hardware Virtual Machine for the Networked Reconfiguration. In *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 194–197, 2000.

[102] M. Sabeghi and K.L.M. Bertels. Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach. In *Design, Automation and Test in Europe (DATE)*, pages 1576–1579, 2009.

[103] W. Fornaciari and V. Piuri. General Methodologies to Virtualize FPGAs in HW/SW Systems. In *Proceedings of the 1998 Midwest Symposium on Systems and Circuits*, pages 90–93, 1998.

[104] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Virtualizing and Sharing Reconfigurable Resources in High-Performance Reconfigurable Computing Systems. In *the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, pages 1–8, 2008.

[105] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3):171–200, 2005.

[106] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor - A Distributed Job Scheduler. *Beowulf Cluster Computing with Linux*, pages 307–350, 2002.

[107] S. Wong and F. Anjam. The Delft Reconfigurable VLIW Processor. In *17th International Conference on Advanced Computing and Communications (ADCOM)*, pages 244–251, 2009.

[108] Web page. The OpenCores Project Online, Last Checked: 23-03-2013. `http://opencores.org/projects`.

[109] K. Albayraktaroglu et al. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–9, 2005.

[110] D.G. Higgins, and P.M. Sharp. CLUSTAL: A Package for Performing Multiple Sequence Alignment on a Microcomputer. *Gene*, 73:237–244, 1988.

[111] R. Meeuws et al. A Quantitative Prediction Model for Hardware/Software Partitioning. In *International Conference on Field Programmable Logic and Applications(FPL)*, pages 735–739, 2007.

[112] K. Krauter, R. Buyya, and M. Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Software Practice and Experience*, 32:135–164, 2002.

[113] R. Buyya, D. Abramson, and J. Giddy. Grid Resource Management, Scheduling and Computational Economy. 2000.

[114] J. Huo et al. A Study on Distributed Resource Information Service in Grid System. *The IEEE 36th Annual Computer Software and Applications Conference*, 1:613–618, 2007.

[115] D. Puppin et al. A Grid Information Service Based on Peer-to-Peer. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 454–464. Springer-Verlag, 2005.

[116] Q. Zhang and Z. Li. Design of Grid Resource Management System Based on Information Service. *JCP*, 5(5):687–694, 2010.

[117] Z. Balaton, G. Gombas, and Z. Nemeth. A Novel Architecture for Grid Information Systems. In *the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 274, May 2002.

[118] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 181 –194, 2001.

[119] A. Auyoung, B.N. Chun, A.C. Snoeren, and A. Vahdat. Resource Allocation in Federated Distributed Computing Infrastructures, 2004.

[120] Z. Pohl and M. Tichy. Resource Management for the Heterogeneous Arrays of Hardware Accelerators. *International Conference on Field Programmable Logic and Applications*, 0:486–489, 2011.

[121] S. Wong, T.V. As, and G. Brown. ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT)*, 2008.

[122] G. Marsaglia and W.W. Tsang. The Ziggurat Method for Generating Random Variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[123] G. Marsaglia and W.W. Tsang. A Simple Method for Generating Gamma Variables. *ACM Transactions on Mathematical Software*, 26(3):363–372, 2000.

[124] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing. *Concurrency and Computation: Practice and Experience*, 14:1507–1542, 2002.

[125] R. Wolski, J. Brevik, J.S. Plank, and T. Bryan. Grid Resource Allocation and Control Using Computational Economies. In *Grid Computing: Making the Global Infrastructure a Reality*, pages 747–772. John Wiley and Sons, 2003.

[126] R. Buyya, D. Abramson, and S. Venugopal. The Grid Economy. *Proceedings of the IEEE*, 93(3):698–714, 2005.

[127] B. Pourebrahimi, K. Bertels, G. Kandru, and S. Vassiliadis. Market-based Resource Allocation in Grids. In *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing*, page 80. IEEE Computer Society, 2006.

List of Publications

International Journal

1. **M.F. Nadeem**, I. Ashraf, S.A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **DReAMSim: Evaluation of Task Scheduling Algorithms for Distributed Reconfigurable Systems**, *to be submitted.*

International Conferences

1. **M.F. Nadeem**, M. Nadeem, and S. Wong. **On Virtualization of Reconfigurable Hardware in Distributed Systems**. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW 2012)*, pp. 9, Pittsburgh, PA, USA, September 2012.

2. **M.F. Nadeem**, I. Ashraf, S.A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements**. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2012)*, Shanghai, China, May 2012.

3. **M.F. Nadeem**, S.A. Ostadzadeh, M. Nadeem, S. Wong, and K.L.M. Bertels. **A Simulation Framework for Reconfigurable Processors in Large-scale Distributed Systems**. In *the Proceedings of the 42nd International Conference on Parallel Processing workshops (ICPPW 2011)*, Taiwan, September 2011.

4. **M.F. Nadeem**, S.A. Ostadzadeh, S. Wong, and K.L.M. Bertels. **Task Scheduling Strategies for Dynamic Reconfigurable Processors in Distributed Systems**. In *Proceedings of the 2011 International Conference on High Performance Computing & Simulation (HPCS 2011)*, pp. 8, Istanbul, Turkey, July 2011.

5. M. Shahsavari, **M.F. Nadeem**, S.A. Ostadzadeh, Z. Al-Ars, and K.L.M. Bertels. **Task Scheduling Policies for Reconfigurable Distributed Systems: A Survey and Possibilities**. In *Proceedings of the 22th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2011)*, Veldhoven, The Netherlands, November 2011.

6. **M.F. Nadeem**, S.A. Ostadzadeh, M. Ahmadi, M. Nadeem, and S. Wong. **A Novel Dynamic Task Scheduling Algorithm for Grid Networks with Reconfigurable Processors**. In *Proceedings of the 5th HiPEAC Workshop on Reconfigurable Computing (WRC 2011)*. In *conjunction with the 6th International Conference on High Performance and Embedded Architectures and Compiles (HiPEAC 2011)*, Heraklion, Crete, GREECE, January 2011.

7. M. Nadeem, S. Wong, G. Kuzmanov, M. Shabbir, F. Anjam, and **M.F. Nadeem**. **Low-power, High-throughput Deblocking Filter for H.264/AVC, International Symposium on System-on-Chip (SoC 2010)**. pp. 93-98, Tampere, Finland, September 2010.

8. M. Ahmadi, **M.F. Nadeem**, and S. Wong. **Towards the Performance Analysis of Reconfigurable Hardwares in Grid Networks**. In *Proceedings of the 23rd Canadian Conference on Electrical and Computer Engineering (CCECE 2010)*, pp. 6, Calgary, Canada, May 2010.

9. F. Anjam, S. Wong, and **M.F. Nadeem**. **A Shared Reconfigurable VLIW Multiprocessor System**. In *the 24th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2010)*, pp. 1-8, Atlanta, Georgia, USA, April 2010.

10. S. Wong, F. Anjam, and **M.F. Nadeem**. **Dynamically Reconfigurable Register File for a Softcore VLIW Processor**. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2010)*, pp. 969-972, Dresden, Germany, March 2010.

11. F. Anjam, S. Wong, and **M.F. Nadeem**. **A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors**. In *Proceedings of International Conference on Field Programmable Technology (FPT 2010)*, pp. 403-408, Beijing, China, December 2010.

12. **M. F. Nadeem**, M. Ahmadi, M. Nadeem, and S. Wong. **Modeling and Simulation of Reconfigurable Processors in Grid Networks**. In *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig 2010)*, Cancun, Mexico, December 2010.

13. **M. F. Nadeem**, F. Anjam, S.A. Ostadzadeh, M. Ahmadi, and S. Wong. **Towards the Utilization of Reconfigurable Processors**

in Grid Networks. In *Proceedings of the 21th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2010)*, Veldhoven, The Netherlands, November 2010.

Technical Reports

1. **M.F. Nadeem**, F. Anjam, and S. Wong. **An Overview of Grid Softwares and Applications to exploit the added performance of Reconfigurable Hardwares in Grid Networks**. Technical report, Delft University of Technology, Netherlands, January 2009.

2. F. Anjam, **M.F. Nadeem**, and S. Wong. **Design Methodologies Incorporating Reconfigurable Computing in Grid Networks**. Technical report, Delft University of Technology, Netherlands, December 2009.

Samenvatting

RECENTE vooruitgang in verwerkingssnelheden, netwerk bandbreedte, en middleware technologieën hebben bijgedragen aan nieuwe computerplatforms, variërend van grootschalige computing-clusters tot wereldwijde gedistribueerde systemen. Als gevolg hebben de meeste huidige computingsystemen verschillende types van heterogene verwerkingsresources. Bij het binnengaan van de peta-schaal computing-tijdperk en verder zullen herconfigureerbare verwerking elementen zoals Field Programmable Gate Arrays (FPGA's), als mede toekomstige geïntegreerde hybride computing-cores, een leidende rol spelen bij het ontwerp van toekomstige gedistribueerde systemen. Daarom is het essentieel om simulatietools te ontwikkelen om de prestaties van herconfigureerbare processors in de huidige en toekomstige gedistribueerde systemen te meten. In dit proefschrift, stellen we het ontwerp van een simulatie framework voor om de prestaties van herconfigureerbare processoren in gedistribueerde systemen te onderzoeken. Het framework omvat gedeeltelijke herconfigureerbare functionaliteit van herconfigureerbare knooppunten. Afhankelijk van het beschikbare herconfigureerbare gebied kan elk knooppunt meer dan één taak tegelijk uitvoeren. Als onderdeel van de implementatie van het framework beschrijven we een eenvoudig mechanisme voor de resource onderhoudsinformatie. Wij stellen het ontwerp van datastructuren voor die essentiële onderdelen vormen van een Resource Information System (RIS). Een gedetailleerd voorbeeld is verstrekt om de basisfunctionaliteit van deze data structuren te beschrijven, die de informatie onderhoudt betreffende de reconfigureerbare knooppunten, zoals hun bijgewerkte statussen, hun beschikbare gebied, de huidige taken, enz. Bovendien presenteren we als casestudy een scala aan scheduling strategieën geïmplementeerd om taken over herconfigureerbare verwerkingsknooppunten te verdelen, door gebruik te maken van de mogelijkheid van gedeeltelijke en volledige herconfigureerbaarheid van de knooppunten. Wij stellen een generieke scheduling algoritme voor dat taken kan toewijzen aan deze twee knooppunten varianten. Door gebruik te maken van een bepaalde set van simulatie parameters, onder dezelfde simulatie voorwaarden, hebben we verschillende simulatie-experimenten uitgevoerd. Op basis van de resultaten wordt aangetoond dat knooppunten met gedeeltelijke herconfigureerbare opties minder gemiddelde wachttijd per taak hebben en totale taak doorlooptijd. Bovendien suggereren de resultaten dat de gemiddelde verspilde oppervlakte per taak minder is vergeleken met de volledige configuratie, wat de functionaliteit van de simulatie framework verifieert.

Propositions

accompanying the PhD dissertation

Evaluation Framework for Task Scheduling Algorithms in Distributed Reconfigurable Systems

by M. Faisal Nadeem

1. The future of FPGA technology in HPC domain lies in its notion of reconfigurability.
2. GPUs will quickly marginalize FPGAs in HPC, unless the FPGA vendors offer standard open-source platforms.
3. If FPGA technology wants to survive in a competitive HPC computing environment, the designers/vendors should pacify the partial reconfigurability.
4. Conflict between Science and Religion is unavoidable.
5. The louder the voice, the weaker the argument.
6. All the deterministic actions in the universe are the result of accumulated randomness.
7. Social media bring distant people closer, and vice versa.
8. Too much of rationality leads to atheism.
9. It is difficult to handle failure, but it even more difficult to handle success.
10. Don't ask me, but ask Google: because I will also do so.
11. The European Union model must be followed by more global regions for peace and prosperity.
12. Cricket has glued together more Pakistanis than anything else.
13. Murphy's law of PhD defense: When you are ready for more technical questions, you get more general questions.
14. Everything in nature is lyrical in its ideal essence, tragic in its fate, and comic in its existence. (George Santayana)
15. Every society has the criminals it deserves. (Emma Goldman)
16. A belief which leaves no place for doubt is not a belief; it is a superstition. (Jose Bergamin)

These propositions are regarded as opposable and defendable, and have been approved as such by the promoter, Prof.dr. K.L.M. Bertels.

Stellingen
behorende bij het proefschrift

Evaluation Framework for Task Scheduling Algorithms in Distributed Reconfigurable Systems

door M. Faisal Nadeem

1. De toekomst van FPGA technologie in het HPC domein ligt in de notie van reconfigureerbaarheid.
2. GPUs zullen FPGAs in HPC snel marginaliseren, tenzij FPGA leveranciers standaard opensource platforms leveren.
3. Als FPGA-technologie wil overleven in een concurrerende HPC rekenomgeving, moeten ontwerpers/verkopers gedeeltelijke reconfigureerbaarheid pacificeren.
4. Conflict tussen Wetenschap en Religie is onvermijdbaar.
5. Hoe luider de stem, des te zwakker het argument
6. Alle deterministische acties in het universum zijn het gevolg van geaccumuleerde willekeurigheid.
7. Sociale media brengen verre mensen dichterbij, en vice versa.
8. Te veel van rationaliteit leidt tot atheïsme
9. Het is moeilijk om met mislukking om te gaan, maar het is nog moeilijker om met succes om te gaan.
10. Vraag het niet aan mij, maar vraag het aan Google: want ik zal dat ook doen.
11. Het model van de Europese Unie moet door meer globale regio's worden gevolgd voor vrede en welvaart.
12. Cricket heeft meer Pakistanen samengebracht dan wat dan ook.
13. Murphy's wet van doctoraatsverdediging: Wanneer je gereed bent voor meer technische vragen krijg je meer algemene vragen.
14. Alles in de natuur is lyrisch in zijn ideale essentie, tragisch in zijn lot, en komisch in zijn bestaan. (George Santayana)
15. Elke samenleving heeft de criminelen die het verdient. (Emma Goldman)
16. Een geloof waar geen plaats voor twijfel is, is geen geloof; het is een bijgeloof. (Jose Bergamin)

Deze stellingen worden opponeerbaar en verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor, Prof.dr. K.L.M. Bertels.

Curriculum Vitae

M. Faisal Nadeem was born on February 10, 1980 in Kohat, Pakistan. He completed his bachelors degree in Electrical Engineering from the NWFP University of Engineering and Technology (UET) Peshawar in 2002. Subsequently, he graduated with MSc in Information Technology (IT) from Pakistan Institute of Engineering and Applied Sciences (PIEAS) Islamabad in 2004. Later, he worked as a Senior Engineer in Pakistan Institute of Science and Technology, Islamabad, from 2004 to 2007.

In December 2007, he obtained a scholarship from Higher Education Commission (HEC) Pakistan for pursuing PhD studies, and joined Computer Engineering (CE) Labs, in EEMCS faculty at Delft University of Technology, the Netherlands. He worked on task scheduling algorithms in reconfigurable distributed systems, simulation frameworks, queuing networks, and adaptable VLIW processors, under the supervision of Dr. J.S.S.M. Wong. The research conducted by him is presented in this thesis. He is a member of HiPEAC, ACM, and IEEE.

His general interests include cricket, sports, reading, social media, writing blogs, music, arts, history, politics, science, and interfaith dialogue.