8-1-2016

A Distributed Processing Platform With Reconfigurable Autonomous Nodes

Grzegorz Chmaj
University of Nevada, Las Vegas

A DISTRIBUTED PROCESSING PLATFORM WITH RECONFIGURABLE AUTONOMOUS NODES

By

Grzegorz Chmaj

Master of Science in Computer Science
Wrocław University of Technology
2005

Doctor of Philosophy – Computer Science
Wrocław University of Technology
2010

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy – Electrical Engineering

Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2016

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

June 28, 2016

This dissertation prepared by

Grzegorz Chmaj

entitled

A Distributed Processing Platform with Reconfigurable Autonomous Nodes

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy – Electrical Engineering
Department of Electrical and Computer Engineering

Henry Selvaraj, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Shahram Latifi, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Examination Committee Member

Laxmi P. Gewali, Ph.D.
Graduate College Faculty Representative

ABSTRACT

Distributed processing is a fast growing area of interest due to the exploding popularity of Internet of Things (IoT) and Unmanned Aerial Vehicles (UAV) technologies. IoT is a distributed processing structure by nature, while UAVs evolve from single-UAV applications towards multiple-UAV (teams). The demand for processing capabilities is expanding as well. The general purpose processors (e.g. CPUs) can be used for any type of application, however this flexibility is at the cost of operational efficiency. Application Specific Integrated Circuits (ASICs) are designed for certain types of application and have great operational efficiency, but they rarely can be used for other applications. The reconfigurable chips – Field Programmable Gate Arrays (FPGAs) provide high operational efficiency along with the application flexibility – as they can be reprogrammed with the functionality that is required at the given time. All the above listed aspects are combined in the distributed processing system that is expected to consume low amount of electrical energy.

This dissertation proposes a comprehensive solution for the problem of distributed processing equipped with reconfigurable units. The complete and detailed architecture is provided for each element. The design includes operational algorithms that together with the architecture constitute a complete solution for the stated problem. The design of the units is flexible and allows any number and combination of CPUs, ASICs or FPGAs. Units in the proposed design are autonomous – the decisions are taken by individual units, instead of the central node, which is marginalized. The decentralized and autonomous approach provides more flexible and reliable design that is especially important for IoT and teamed UAV applications. The efficiency of the proposed solutions is defined as electrical energy consumption and operation timespan, and is measured using dedicated experimentation system through numerous simulations.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor Henry Selvaraj, my research supervisor, for his guidance, constructive suggestions, invaluable advices, useful critiques and overall support. Dr. Selvaraj's assistance has been a great help and was leading this dissertation research in the proper direction.

I wish to acknowledge the cooperation with Dr. Shahram Latifi, Dr. Emma Regentova and Dr. Laxmi Gewali, who provided very useful feedback during various stages of the doctoral process.

Finally, I would like to thank Dr. Dawid Zydek for numerous advices and continuous support.

TABLE OF CONTENTS

Abstract iii
Acknowledgements iv
List of Tables vii
List of Figures viii
List of Algorithms xi
List of Symbols xii
Chapter 1 Introduction 1
Chapter 2 Literature Overview 8
Chapter 3 Thesis 18
Contributions 20
Chapter 4 Reconfigurable system architecture - DPRS 21
4.1. Proposed notation 21
4.2. Overview of the DPRS 21
Layered architecture 24
Object architecture: 24
4.3. System design and operation 25
Interfaces 25
Time scale 26
DPRS structure 27
Incentives 57
Fully decentralized systems (DHT) 57
4.4. Operation of DPRS 59
4.3.1. General operation of the DPRS 59
4.5. Operational algorithms 74
4.5.1. AL_RECONFIGURE_FPGA 74
4.5.2. AL_MATCH_BLOCK_TO_PU 78
4.5.3. Other algorithms 91
4.6. Other mechanisms 91
The weak-node exclusion mechanism 91
Step function 92
Chapter 5 Experimental Study and Analysis 93
5.1. The experimentation plan 93
Energy utilization 94

Task execution time 98
Utilization rates 98
5.2. The evaluation environment 98
5.3. Energy utilization 102
5.4. Experiments 102
5.4.1. Impact of share of FPGAs 102
5.4.2. The impact of AL_RECONFIGURE_FPGA 104
5.4.3. Enabling the reconfiguration with AL_ALLOW_RECONFIGURE 109
5.4.4. Reconfiguration period 110
5.4.5. Impact of MATCH_BLOCK_TO_PU 120
5.4.6. Summary of efficiency gain 134
Chapter 6 Conclusions 138
Appendix 140
References 147
Curriculum Vitae 155

LIST OF TABLES

Table 1. Basic roles in the DPRS system 31
Table 2. Energy expenditure elements 94
Table 3. Energy modeling parameters 96
Table 4. Configurations used for Fig. 50 – Fig. 53 112
Table 5. Properties of applications used 122
Table. 6. Comparison of AL_MATCH_BLOCK_TO_PU_3 to remaining algorithms (E_t) 125
Table. 7. Comparison of AL_MATCH_BLOCK_TO_PU_3 to remaining algorithms (E_p) 125
Table. 8. The utilization of the processing units 132
Table. 9. The utilization of the processing units (averaged values) 133
Table 10. The distribution of the expenditure elements. 137

LIST OF FIGURES

Fig. 1. Logical view for the distributed processing system 1
Fig. 2. The idea of task dividing (with single task manager) 2
Fig. 3. Task division and result merge 3
Fig. 4. Time to market for FPGA & SoC and ASSP / ASIC 4
Fig. 5. High level view of the common distributed processing 5
Fig. 6. High level view of the proposed system 22
Fig. 7. Layered architecture of proposed solution 24
Fig. 8. Structure of IIS 26
Fig. 9. DPRS architecture 27
Fig. 10. Architecture of *Node* object 28
Fig. 11. Control nodes 31
Fig. 12. Architecture of *Function* object 32
Fig. 13. Architecture of the *Function Implementation* object 32
Fig. 14. Architecture of the *datasource* object 33
Fig. 15. Architecture of the *block* object 34
Fig. 16. Workflow for blocks defined and blocks offline 37
Fig. 17. Workflow for block cooperative 38
Fig. 18. Workflow for block online 39
Fig. 19. Architecture of *result* object 41
Fig. 20. Architecture of *Task* object 42
Fig. 21. Task status flow 45
Fig. 22. Architecture of *processing unit* object 46
Fig. 23. Workflow for non-reconfigurable processing unit 48
Fig. 24. Workflow for reconfigurable processing unit 49
Fig. 25. Idea of communication layer 50
Fig. 26. Architecture of *message* object 51
Fig. 27. Architecture of *nodeAssets* object 56
Fig. 28. General workflow of DPRS system for a single task 61
Fig. 29. Top-level scheme for node phases (AL_NODE) 63
Fig. 30. Operation of ROLE_BASIC 64
Fig. 31. Node workflow for ROLE_CONTROL 66
Fig. 32. ROLE_PROCESSING processing stages 69

Fig. 33. Reconfiguration scheme (ROLE_RECONFIGURABLE) 70
Fig. 34. Operation of ROLE_TASK_OWNER 73
Fig. 35. The operation of AL_RECONFIGURE_FPGA_1 75
Fig. 36. The operation of AL_RECONFIGURE_FPGA_2 76
Fig. 37. Operation of AL_RECONFIGURE_FPGA_3 78
Fig. 38. Operation of AL_MATCH_BLOCK_TO_PU_1 algorithm 80
Fig. 39. Operation of AL_MATCH_BLOCK_TO_PU_2 algorithm 82
Fig. 40. Operation of AL_MATCH_BLOCK_TO_PU_3 algorithm 85
Fig. 41. Operation of AL_MATCH_BLOCK_TO_PU_4 algorithm 88
Fig. 42. Operation of AL_MATCH_BLOCK_TO_PU_5 algorithm 90
Fig. 43. Cost per % of FPGAs and S_{thres} illustration 103
Fig. 44. FPGA reconfiguration algorithms impact to E 105
Fig. 45. Communication costs E_t 106
Fig. 46. Computation costs Ep 106
Fig. 47. Values of C for AL_RECONFIGURE_FPGA algorithms 107
Fig. 48. Processing time required for AL_RECONFIGURE_FPGA algorithms 108
Fig. 49. Reconfiguration on/off 109
Fig. 50. Total cost as a function of reconfiguration period (configuration 1) 112
Fig. 51. Total cost as a function of reconfiguration period (configuration 2) 113
Fig. 52. Total cost as a function of reconfiguration period (configuration 3) 113
Fig. 53. Total cost as a function of reconfiguration period (configuration 4) 114
Fig. 54. RECONFIGURATION_PERIOD prediction 118
Fig. 55. *RP_ABS* for investigated systems 119
Fig. 56. E for $Q = 0.5$ 121
Fig. 57. Communication energy for $Q = 0.5$ 122
Fig. 58. Processing energy for $Q = 0.5$ 123
Fig. 59. Execution time for system with $Q = 0.8$ 124
Fig. 60. Relation of Q to E, application *5x5t1* 126
Fig. 61. Relation of Q to E, application *5x5t2* 126
Fig. 62. Relation of Q to E, application *5x5t3* 127
Fig. 63. Relation of Q to E, application *5x5t4* 127
Fig. 64. Relation of Q to E, application *5x5t5* 128
Fig. 65. Relation of Q to processing time, application *5x5t* 129
Fig. 66. Relation of Q to processing time, application *5x5t2* 129
Fig. 67. Relation of Q to processing time, application *5x5t3* 130

Fig. 68. Relation of *Q* to processing time, application *5x5t4* 130
Fig. 69. Relation of *Q* to processing time, application *5x5t5* 131
Fig. 70. Usage of processing units, AL_MATCH_BLOCK_TO_PU_1 133
Fig. 71. Usage of processing units, AL_MATCH_BLOCK_TO_PU_3 134
Fig. 72. Distribution of operations for the non-optimized case 136
Fig. 73. Distribution of operations for the optimized case 136
Fig. 74. Gantt graph: System operation for AL_MATCH_BLOCK_TO_NODE_1 and AL_MATCH_BLOCK_TO_NODE_3 143
Fig. 75. Gantt graph: system behavior for non-optimized and optimized operations 146

LIST OF ALGORITHMS

Algorithm: AL_RECONFIGURE_FPGA_1 74
Algorithm: AL_RECONFIGURE_FPGA_2 75
Algorithm: AL_RECONFIGURE_FPGA_3 77
Algorithm: AL_MATCH_BLOCK_TO_PU_1 79
Algorithm: AL_MATCH_BLOCK_TO_PU_2 81
Algorithm: AL_MATCH_BLOCK_TO_PU_3 83
Algorithm: AL_MATCH_BLOCK_TO_PU_4 86
Algorithm: AL_MATCH_BLOCK_TO_PU_5 89
Algorithm: AL_DESIGNATE_CONTROL_ROLES 91
Algorithm: AL_REASSIGN_ROLES 91
Algorithm: AL_ALLOW_RECONF 91
Weak node exclusion 91

LIST OF SYMBOLS

A – interfaces: a = 1, 2, …, A

B – blocks in the system: b = 1, 2, …, B

B_{z1} – number of *blocks defined* for task z

B_{z2} – number of *blocks online* for task z

B_{z3} – number of *blocks cooperative* for task z

B_{z4} – number of *blocks offline* for task z

BU – blocks unallocated / blocks

C_{E1}^{E2} – comparison of the values of E2 and E1

D – datasources: d = 1, 2, …, D

E – energy spent on the operation

F – functions: f = 1, 2, …, F

FP – number of functions programmed

FS – functions programmed

G – sockets: g = 1, 2, …, G

H – links: h = 1, 2, …, H

H' – step discrete function

M_f – size of the block data that is related to function f

ND – array of nodes

O – objects: o = 1, 2, …, O

P – processing units: p = 1, 2, …, P

Q – relation of efficient to inefficient nodes

R – roles defined in the system: r = 1, 2, …, R

$RPO_{\Omega\varepsilon}$ – optimal value of RECONFIGURATION_PERIOD for the system Ω at tolerance ε

RP_{acc} – accuracy of setting *RECONFIGURATION_PERIOD* against optimal value $RPO_{\Omega\varepsilon}$

RP_E – energy consumed in function of reconfiguration period accuracy

RP_ABS – absolute impact of the reconfiguration period setting

S – FPGA to CPU ratio

T – time slots modeling the timespan of the system operation: $t = 1, 2, \ldots, T$

V – nodes present in the system: $v = 1, 2, \ldots, V$

Z – tasks present in the system: $z = 1, 2, \ldots, Z$

Ω – the DPRS system

Ψ_o – size of the object *o* (kB)

Φ_f – occurrences of *f*

$\lambda_{b,f}$ – function describing the threshold of functions required by a block *b*

θ_Ω – minimum of the *E* for reconfiguration period evaluation for Ω DPRS

$\theta_{\Omega M}$ – local minimum of the *E* for reconfiguration period evaluation for Ω DPRS

a – interface

b – block

b_{t1} – indicator: if block *b* is *block defined* ($b_{t1} = 1$; 0 otherwise)

b_{t2} – indicator: if block *b* is *block online* ($b_{t2} = 1$; 0 otherwise)

b_{t3} – indicator: if block *b* is *block cooperative* ($b_{t3} = 1$; 0 otherwise)

b_{t4} – indicator: if block *b* is *block offline* ($b_{t4} = 1$; 0 otherwise)

b_L – indicator: if block *b* requires datasource values to be local ($b_L = 1$; 0 otherwise)

c_p – processing power of processing unit *p*

h_v – indicator: link h belongs to node v

h_{v1} – indicator: link h is used

i – function implementations: $i = 1, 2, \ldots, F$

$k_{v,d}$ – indicator: if datasource d is present on node v ($k_{v,d} = 1$, 0 otherwise)

$l_{b,d}$ – indicator: if datasource d is required by block b ($l_{b,d} = 1$, 0 otherwise)

$m_{p,type}$ – indicator: if processing unit p is the type of *type*

$n_{b,f}$ – indicator: if block b requires function f ($n_{b,f} = 1$; 0 otherwise)

$q_{v,r}$ – indicator: if role r is assigned to node v ($q_{v,r} = 1$; 0 otherwise)

$u_{p,g}$ – indicator: if processing unit p is installed in socket g ($u_{p,g} = 1$; 0 otherwise)

v_{up} – upload bandwidth for node v

v_{dn} – download bandwidth for node v

w – node, additional index along with v

$w_{v,g}$ – indicator: if socket g is installed on node v ($w_{v,g} = 1$; 0 otherwise)

$x_{b,z}$ – indicator: if block b belongs to task z ($x_{b,z} = 1$; 0 otherwise)

CHAPTER 1

INTRODUCTION

Distributed processing system is used for processing a task over a distributed structure of participants, each contributing its processing power. Such a system comprises of multiple devices (called *nodes*) equipped with processing capabilities (CPU, ASIC, DSP, etc.) and communication links providing the network communication. Nodes join the designated system, become its members and contribute with their processing power and any other capabilities they possibly might have – such as connected peripherals. Elements of the system – control element and member nodes – communicate over a common network layer. All these together constitute a distributed processing system. This kind of a system is capable of processing a task by internally splitting it into smaller parts (blocks/chunks) that are independently processed on separate nodes, and results are then sent to the control unit where they are combined to provide the final result (Fig. 2). The input task is provided to the control unit, and the final result is also produced by the control unit – therefore the distributed processing system can be interpreted as a single virtual machine in which all the underlying computation, communication and networking are blackboxed (Fig. 1).

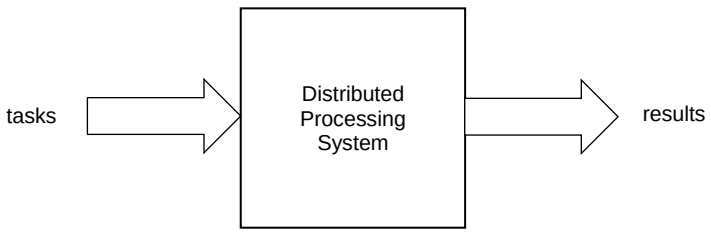


Fig. 1. Logical view for the distributed processing system

Distributed processing systems can process tasks of various types. Popular distributed processing systems such as public computation systems [And04], [Boi09], [Chm10] and grids [Buy02], [KBM02], [TMK06] are usually used for computing (e.g. signal processing, image analysis, solving heavy computational problems, etc.) – and the result is either the computational result (Fig. 3) or some answer (e.g. "object found in the processed image"). Due to the evolution of the distributed processing systems, a new task type must be introduced – a type that produces the result that immediately affects the system by modifying its current operating parameters.

Grids consist of multiple machines having large computational power, connected to one logical structure used to process complex tasks, often built by academic institutions. Multiple assets can be shared: computational power, disk space, data etc. Public computation systems consist of home/office machines connected over the network, so the hardware investment becomes redundant, unlike in grids. Both grids and public computation systems are typically centralized – designated unit manages all the processing, source data and processing the results.

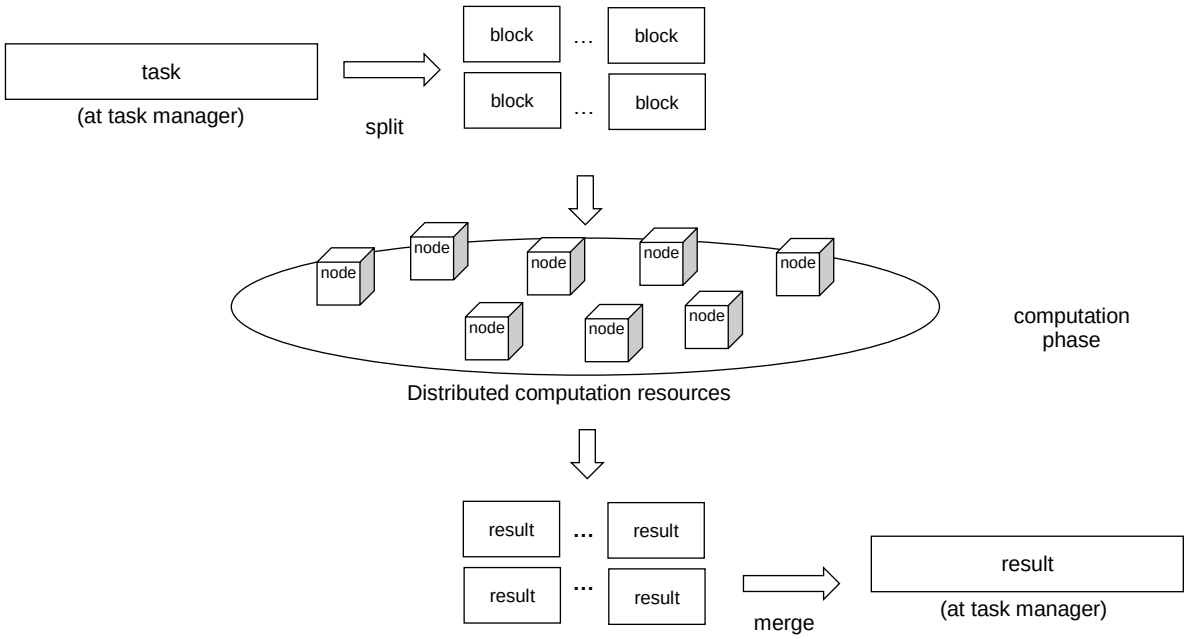


Fig. 2. The idea of task dividing (with single task manager)

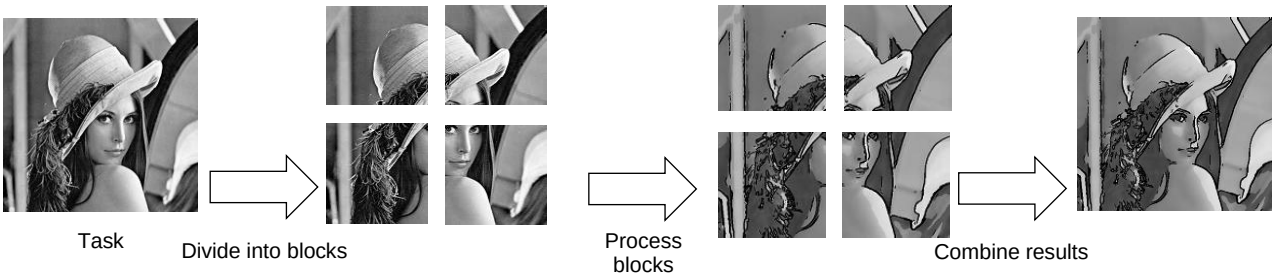


Fig. 3. Task division and result merge

Reconfigurable systems are deployed for those elements that could be reconfigured before the system start, or during the system operation. The comprehensive survey of reconfigurable systems, including concepts, challenges, features and applications is presented in [JN09], and the development methods and tools used for embedded reconfigurable systems are described in [JNF10]. The design patterns for reconfigurable computing are described in [Deh04]. The element that provides the reconfiguration ability is usually an FPGA (Field Programmable Gate Array) – that can be programmed with bitstream. Bitstream is the binary object created as the result of digital logic design and synthesis. In the early FPGA days, the new configuration could be programmed before the FPGA was used. Later, many solutions that allow on-the-fly reconfiguration appeared – and reconfiguration became possible during the system operation. The advantages of FPGA are significant. For a given application, there are few ways to perform its execution. The most common and universal way is to use the CPU chips – they are general purpose devices and their instruction sets can be used to build any function. The efficiency is questionable, compared to other solutions, as CPUs are not designed specifically for any given application. The second approach is based on ASIC chips (Application Specific Integrated Circuits). They are designed to target a specific application (or the set of applications) and contain multiple specialized functions implemented in hardware. This makes them very efficient and fast, however all the functions must be known during the design phase – so possible extension of hardware-

implemented function set is difficult and takes a long time. The financial cost is another aspect to consider – the design and fabrication process requires a lot of resources and effort, and practical implementation of ASIC is reasonable only when manufacturing thousands or more ASICs. The third approach is the use of FPGAs – the application specific functions are first created using either HDL (High Definition Language) or schematic design. Then the synthesis process is performed and the resulting bitstream is created, which is later programmed into FPGA. This process results in a hardware chip, that has application-specific functions implemented – thus the operational efficiency is high, and the whole process is fast (easy to program FPGA when the bitstream is readily available). The possibility of reconfiguring FPGA on-the-fly is the additional profit. The comparison of time to market processes is shown in Fig. 4 [Ola12], and the evolution of FPGAs is described in [PTD13].

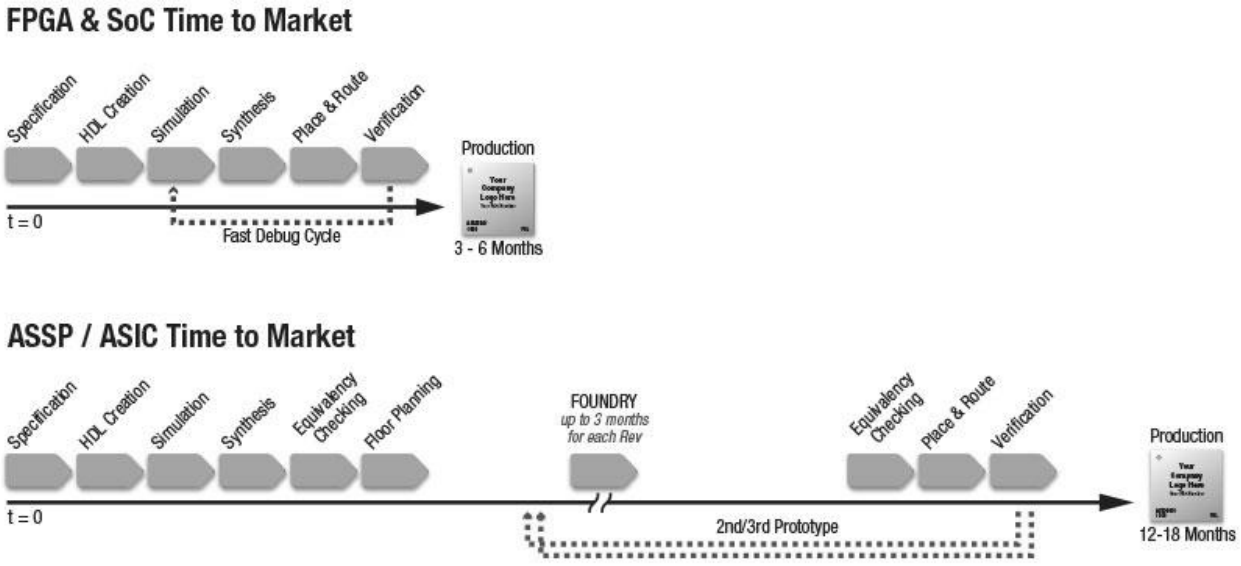


Fig. 4. Time to market for FPGA & SoC and ASSP / ASIC

The structure of a DPS (*Distributed Processing System*) usually consists of: the intercommunication structure (IS); the central element (Fig. 5) performing the control tasks; and machines contributing their computation resources. Also, the command and control is concentrated at the management node – so its roles might be listed as follows:

- manages the tasks that are inputted to the system for processing
- performs the task to subtasks division (subtasks are also called *blocks*)
- assigns the subtasks to the nodes
- controls the result delivery
- manages the results received
- commands the nodes for other purposes.

The common nodes (computational ones) behave pretty much in a static way – they execute commands received from the management node. Usually they only refuse commands only when they lack the resources required to complete the request.

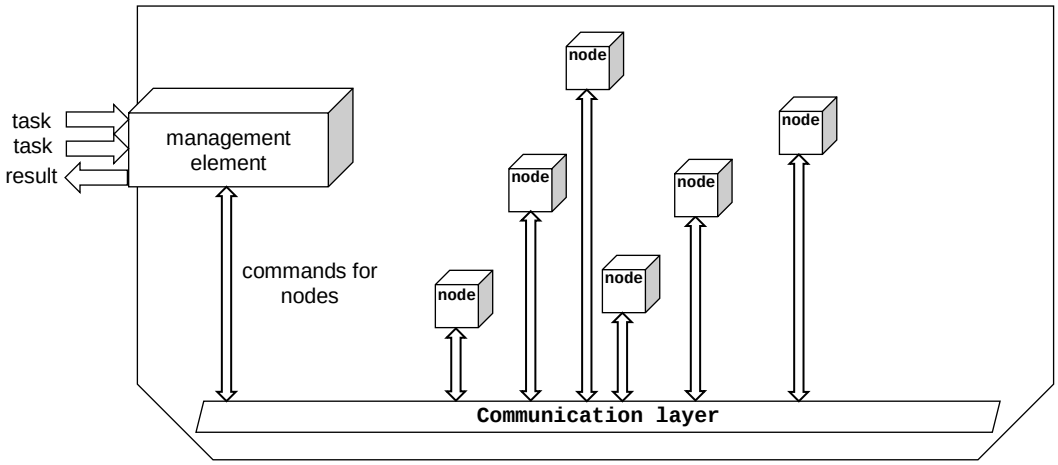


Fig. 5. High level view of the common distributed processing

Centralized architecture is easy to implement and manage. However, having the central element brings in the problem of a single point of error – without the central element the system does not operate. The issues are the same as faced in the peer-to-peer distributed systems serving as media sharing systems, which at some point became the target of intellectual properties agencies. The shutdown of the centralized systems, such as Napster, could be easily achieved, as the only thing required was to shut down the central management element. On the other hand, fully decentralized systems such as Gnutella – has also been implemented. It showed that full decentralization was a very challenging problem. Gnutella used broadcasting participants with system messages that flooded the system and, at some point, was rendering it inoperable due to the inefficiency. Another big problem was the data localization and search – there was no central server that would serve as the location database (as it happens in the centralized systems). To various extents, DPSes suffer from similar problems as do their centralized counterparts. In DPS, additionally, all nodes are active system elements (take many autonomous decisions, contrary to centralized systems where nodes are usually fully controlled). Therefore, the system management in DPS has to be reliable and efficient as a whole. As data at the data sources can change constantly, the data has to be fetched at the time it is needed. The structure of Unmanned Aerial Vehicles (UAVs), where each UAV provides geographic and environment data to other UAVs is an example of such a system. UAVs may be separated from the command center, so the use of a distributed control provides the independency required in this case. UAV-based systems implement many applications operating on multiple nodes, such as: general-purpose distributed processing applications, object detection, tracking, surveillance, data collection, path planning, navigation, collision avoidance, coordination and environmental monitoring. The comprehensive survey of the UAV applications in the distributed systems is done in [CS15]. Another emerging field of the distributed processing is the

Internet of Things (IoT). IoT systems are composed of multiple devices connected to the Internet and gain a broad spectrum of new applications [CS16]. Among those applications, many require huge amounts of processing power typically not available on a single device. It can be assumed that other IoT devices nearby have their processing power available and might be staying in the idle mode at the time. Thus a distributed processing system is the solution that enables the devices to share their resources in the nearby/logical group and reach the defined processing goal. The system described in [GBA12] describes the use of IoT/Cloud computing in healthcare services. Authors have discussed modifications to IoT-based systems to improve the quality of life of people with chronic diseases. Multiple middleware solutions have been proposed [PZC14] to realize the context-aware computing in Internet of Things systems. This kind of middleware enables the services that would run on a given infrastructure, while usually they are defined at the high level. Another popular topic in the area of IoT applications is smart housing. The solutions proposed in the literature mostly do not include energy optimization and are centrally-managed ([KPR13], [PZC14]). The solution of integrating the smart home system with webservices and cloud computing is presented in [SAH13]. [CS16] describes a distributed processing solution for Internet of Things that optimizes the operation in the field of energy consumption with the main focus on communication and computation.

CHAPTER 2

LITERATURE OVERVIEW

Distributed processing systems operate on various scale: starting from System-on-Chip local structures, through multiprocessor structures (local), systems with the nodes located in the vicinity, to the big geographically dispersed structures, with many nodes present on various continents. The author has presented solutions for multiprocessor distributed processing systems [CZE12], public distributed computation systems (geographically spread nodes communicating over the internet) [CWT12], [CW12] and others. The author has also addressed the energy consumption aspects for the reconfigurable systems in [CSG13] and the approach of using the DHT in the distributed computing system with multiple data sources showing the efficiency of the proposed decentralization approach. Distributed systems evolve into more intelligent, self-sufficient, self-aware and autonomous systems [AC15]. The design of modern distributed systems includes the reconfigurability more and more often, still facing many challenges such as power-efficient computation, fault tolerance, targeted systems (embedded systems, high-performance multicores), specific memory architectures and many others [SG11]. The work proposed in this dissertation addresses the power-efficient computation and embedded systems among others. The reconfigurable distributed processing area is considered not only as the research and commercial topic, but also as one of the key technologies to apply in military UAV systems – as listed by the Department of Defense of USA in the roadmap document [Dod13].

A general description of reconfigurable systems, along with the characteristics of reconfigurable logic were described in [BP02]. The fundamental differences between traditional processing architectures and reconfigurable logic were stated (spatial computation, configurable data path, distributed control and distributed resources). Authors also state the architectural parameters: granularity (size of the smallest functional unit addressed by mapping tools), host coupling (coupling with the host processor that determines the cost of the reconfiguration), reconfiguration methodology (what is the process of reconfiguring the FPGA unit) and memory organization (what is the memory access for reconfigurable logic and host). This paper also lists the reconfigurable architectures and the application of vision processing. Authors of [FBP08] present the methods of workload distribution over processing resources, which are considered to be multi-core CPUs, GPUs (Graphical Processing Units), PPU (Physics Processing Unit) and FPGAs, among others. Their work focuses on handling applications' requirements in the high-level design process. The concurrencies are identified and used to achieve load balancing, providing an efficient task distribution. Therefore, the reconfiguration process in [FBP08] concerns the task-to-node distribution/assignment (tasks migration) with regards to defined constraints such as timing, performance, precision and others. First assignment of tasks to nodes is offered to be done two different ways: a) the optimal distribution using ILP (Integer Linear Programming) b) non-optimal distribution based on the estimation. After the initial assignment is done, the run-time reconfiguration algorithm is used to periodically attempt to rearrange the tasks distribution. A surveillance system based on UAVs is presented as a case study of the presented idea. The use of reconfigurability mechanisms for wireless sensor networks was presented in [HWH13]. Authors show the reconfigurable network of wireless sensor nodes that can be used for multiple applications, and retain low energy usage. Nodes in the presented system are reconfigurable and

are managed to take applications' requirements into account. Tasks are distributed to the nodes in a way to meet the QoS and minimize energy consumption. The term *accelerator* is used to model the preferable configuration for application, and each node can accommodate a specified number of accelerators in its fabric. Reconfiguration energy is considered, and each application has the preferable accelerators list, memory requirement and QoS requirement. Nodes reconfigure themselves implementing the list of accelerators, to match the needs of applications executed over WSN. The reconfigurable heterogeneous multi-core System-on-Chip is presented in [KK08], where the computation-intensive applications are offloaded to the reconfigurable FPGAs to increase efficiency. System also contains general purpose CPUs and dedicated chips such as DSP. Processors use the pool of reconfigurable units to offload some tasks. The focus of the research is put on the communication network, system structure and messaging details, however the scheduling algorithms and resource management are omitted.

The framework for distributed processing for dishonest nodes was presented in [Roz04]. The author considers a situation, where distributed system contains nodes that want to alter the results. Multiple metrics are defined and the computation results produced by each node are judged by designated judges. Open Control Platform for reconfigurable distributed control systems was presented in [WSK00]. It provides the coordination of distributed interaction among hierarchically organized units along with the support of their dynamic reconfiguration. The low-level implementation details are separated (abstracted) from the level of standardized interfaces. The communication is based on widely-used CORBA standard, providing the real-time communication mechanisms between distributed objects. Nodes implement the architecture of signals and ports and therefore, connections between ports and signals can be dynamically created during runtime, using reconfiguration. Mapping tasks or other demands on the distributed nodes may occur during

runtime, but also can happen at earlier stages. A framework for simulating the behavior of reconfigurable nodes in distributed systems is presented in [NON11]. It includes modeling complex reconfigurable nodes, processor configurations and tasks along with general purpose processors. The framework implements the following performance metrics: average waiting time per task, average running time of each task, average reconfiguration count per node, average reconfiguration time per task, average wasted area per task, average scheduling steps per task, total discarded tasks, total scheduler workload, total used nodes and total simulation time. Each reconfigurable node is configured with a given configuration, and the resource management system (RMS) is included in the system. RMS can send configurations to the nodes individually, it also contains a scheduling module that takes user tasks and maps them onto the nodes – that is, the task-node assignment is controlled in a centralized manner. Tasks are also described with requirements: each task requires a processor of a given configuration with specific parameters. Authors also demonstrate an example of the algorithm implemented in the framework and present the results of the experiments. Framework provides multiple techniques for analyzing the node reconfigurability in the distributed systems, and is yet another example of the simulator for distributed processing, designed to handle specific cases when there is a lack of a universal simulation framework. Authors extended their simulation framework in [NAO12] with the partial node reconfigurability – part of the node can be configured with a new functionality, while the remaining runs the previously loaded configurations. This approach resembles the partial FPGA reconfiguration, also – the reconfiguration of the node can be performed during runtime. Authors also added the data structure mechanism allowing the retention of dynamic statuses of the nodes. On a top level, simulator contains four subsystems: input subsystem (allowing user inputs to the framework, set up the specifications, generates tasks, creates configurations); information

subsystem (provides resource information during the simulation, contains: job submission manager (simulates task arrivals) and resource information manager (maintains information about nodes, such as reconfigurable area, family type, current set of processor configurations, state, number of running tasks etc.)); core subsystem (consists of task scheduling manager, monitoring module and load balancing module); output subsystem (gathers the results and generates the simulation report).

Another approach to the middleware is presented in [KKG12], where a distributed system containing heterogeneous reconfigurable hardware units on chip is described, and the method of virtualizing the communication for this system is presented. It provides access to the distant operating system services, inter-process communication, implementation and mapping abstraction – thus the application running without the knowledge of how and where its processes are executed. Proposed architecture consists of the following elements: a) a set of general purpose processors that can support the execution of OS services, GPPs not necessarily are homogeneous; b) a set of dynamically reconfigurable areas, that are responsible for executing the hardware tasks – OS services running in the hardware c) global communication platform between the hardware and the software, based on the dedicated network on chip (NoC). The proposed middleware brings in a lot of flexibility, as it provides a single interface to the lower hardware layers. The framework providing the abstraction over the heterogeneous computational resources is shown in [TPT12]. The reconfiguration is supported, however, the reconfiguration is about placing the microprocessor in the FPGA fabric, and separate hardware accelerators implementing DSP are available. The framework defines the self-adaptive network elements to implement work distribution and resource management. The framework is centralized using the root element that delegates the tasks

to network elements. Multiple nodes for one application are allowed to enable parallel operation, and multiple applications can execute concurrently.

Another problem widely researched in literature is allocation and scheduling of resources to computing nodes. In [KNP01] a multi-objective resource management system is presented – modeled as a set of discrete, feasible solutions to the problem and the n-dimensional objective function (that is either to be minimized or maximized). This mechanism also allows the user to include his preferences that would later impact the scheduling and allocation result. Another approach to scheduling in reconfigurable embedded systems is presented in [SWP04]. Authors propose the concept of the operating system that provides minimal programming model and a runtime procedures, in which the resource management is done during realtime. The online scheduling with the guarantee of scheduling of hard real-time tasks is provided, along with the runtime overhead evaluation using proposed heuristic algorithms. Regarding tasks allocation, both 1D and 2D resource model cases are included. A related work, presenting the execution environment (application data distribution model, implementing reconfigurable computing in multi-threaded environment, and scheduling techniques) is presented in [FC05].

Reconfigurable systems are often present as System-on-Chip, as presented in [SS09]. The design includes sets of RSoCs (colonies) and each of them knows an Operating System Repository (OSR) node (it's management node) that provides OS along with all its service. The nodes in RSoC can communicate with each other directly or indirectly, as can the OSRs. The algorithm finds the efficient execution configuration for the requested service, and this configuration can be reconfigured during the runtime to adapt to a resource change. Another reconfigurable OS is presented in [AAS12], where heuristic algorithms for the scheduling of hard real time task for partially reconfigurable devices are proposed. The schedulers are described in detail, the nodes

have the ability of FPGA reconfiguration, but the bitstreams are not circulating over the network, they must be available locally on the node. System on chip architectures are of great interest in space applications [Ost08] – described solutions serve for flexible in-flight reconfiguration, authors describe the NoC communication architecture based on ESA SpaceWire interface.

The area that exploits the significant growth in the distributed systems technology is Unmanned Aerial Vehicles – a survey of applications that use distributed systems is described in [CS15], also the military applications and goals are listed in [DoD13]. Unmanned Aerial Vehicles (also called 'drones') are the flying vehicles that do not use any pilot on board. They are either controlled remotely from a ground station by a drone operator, or by control algorithms that run autonomously on board. The importance of making the UAVs fully autonomous is recognized, as described in [Tom12]. Authors of [LZL13] present the communication architecture and protocols for inter-UAV networking. Both centralized and decentralized approaches are analyzed, multilayer solutions and multi-group cases are described as well. It is shown, that ad hoc networks are most appropriate for UAV teams, while the multilayer is good for multiple teams of heterogeneous UAVs. The aspects of the UAV-ground communication are described, including the backbone UAV solution, taking the bandwidth and configuration into the consideration.

One of the distributed processing applications is especially worth mentioning – the use of UAV (Unmanned Aerial Vehicles) in a swarm (team) form to achieve a common goal (mission). Recently, small-sized drones have been attracting the attention of both the research and commercial communities. Industry is attempting to use drones for commercial applications (such as surveillance or package deliveries), while scientific researchers focus on swarming, navigation systems, optimization of onboard operational algorithms, and many others. Multiple drones communicating with each other and being commonly controlled are called *swarms*. UAVs

participating in swarms/teams are usually equipped with processing capabilities and are interconnected using a common communication network. This way, such UAV swarms fulfill the assumptions of distributed processing system – in which drones are the nodes. Example applications of distributed processing in UAVs include: object detection, surveillance, data collection, path planning, navigation, tracking, collision avoidance, coordination, environmental monitoring and many others. The use of FPGA reconfigurable nodes in UAV swarm is presented in [KJ07]. Authors propose to equip UAVs with FPGAs and then migrate tasks across the team, so computational and power resources are shared within the system. A distributed operating system is also proposed – it realizes moving the applications among UAVs to provide equal power usage among nodes and continuous replenishment of fully fueled UAVs into the swarm.

Reconfigurability can also be applied to the network connection architecture. [TH10] describes the dynamically reconfigurable network system for cloud computing. The processes in the network are represented by coupling of disassembled features, therefore the dynamic reconfiguration of the network system contains: feature addition, feature deletion, feature moving and connection change not accompanied by feature change. Each node in the network consists of a) flow identification process that identifies which process is to be applied to the packet received b) processing resource that executes the feature that constitutes the process c) switch feature implementing the flexible connection between features d) node management. The dynamic reconfiguration occurs on the node, which can configure the connection between two nodes or allocate the feature locally on the node.

Authors of [CGP13] show the reconfigurable system of mobile agents, of which each can migrate between nodes and between software and hardware implementations. The migration decision might be either taken by agent itself or by the application in general. Agents realize the

computational tasks and coordinate their decision making to reach the global common goal. The hardware agents are realized using the FPGA units that can be reprogrammed during runtime. The goal of reconfiguration and migration of agents can be overall system performance, energy efficiency or other. The proposed architecture contains three layers: a) application layer where agents related to the application domain are identified, b) communication layer c) reconfigurable hardware layer, where FPGAs and hardware agents are located. A reconfiguration process occurs both on the level of FPGA (reprogramming the hardware FPGAs with bitstream) and on the level of agents (migration between software and hardware). Another approach for agent system based on reconfigurability was introduced in [NW02]. This system uses FPGAs as the reconfiguration units, and allows this process both at the initial stage and during the runtime. The architecture includes conventional processor unit, multiple configuration memories and reconfigurable logic part. Agents are acting as interfaces between the environment and the reconfigurable system, and are placed in the configuration memory. Similar to [CGP13], agents can migrate between hardware and software, or also be present in both those environments at the same time. Reconfigurable system with agents, including the embedded system using the FPGA is presented in [NWE04]. Agents are placed in the reconfigurable hardware and implement the agent-based reconfigurable sensor fusion system. System stores the rules of sensor fusion under specific environmental conditions as a knowledge base. Then appropriate rules can be selected according to current needs or sensor conditions. The reconfiguration functionality is implemented through placing the design information directly in the configuration memory. The external conditions affect the internal configuration of the device.

Reconfigurable units can be used in various applications. MATCH is the compiler that uses Matlab as a base, and automatically maps applications onto the distributed computing structure [Ban00].

Such structure may include embedded processors, FPGAs and digital signal processors. There were two optimization objectives included: a) minimization of the resources usage (e.g. type and number of processing units) under performance constraints, b) maximizing performance under resource constraints. For FPGAs, the Matlab code is converted into VHDL, and for DSP chips – the dedicated compiler is used. The mapping to nodes can be done either automatically (through the process of ILP optimization against optimizing resources and performance), or also can be steered by user – to fine tune the program. After the code for each unit is generated, the results are deployed on hardware units.

CHAPTER 3

THESIS

Distributed systems face multiple design problems, including architectural design, hardware limitations and efficiency. On top of that, energy consumption is the common consideration, especially critical for distributed systems based on mobile nodes or those using limited energy sources (battery power, combustible engines).

The problem of distributed processing with multiple data sources (e.g. sensors, cameras, detectors) can be stated as follows:

- given the structure built with multiple devices of various parameters and connected using the network
- each device, if needed, submits the computational task for the execution
- task can contain parts requiring one-time processing, along with the parts that require constant processing during the runtime
- the structure as a whole collaborates to perform the task processing
- devices can be equipped with data sources, gathering the data of various kind
- current reading from data source might be required for processing of some task parts
- devices might be equipped with FPGA integrated circuits, that can be reconfigured during the runtime by obtaining the bitstream and programming the IC

Methods to achieve the goal:

- design all the properties of the system
- include all the required relations between system components
- design the system architecture
- design the communication layer
- propose the operational algorithms for proper system operation
- build the research system in which all the objects are implemented
- perform the experiments applying tasks to the built system and research the properties and efficiency of proposed solutions.

The work presents solutions for architecture, reliability and efficiency of distributed system. It contains the assumptions for the system, technical details, energy model, functions synthesis and definition of the distributed processing framework. The construction of the proposed architecture is a flexible structure with multiple autonomous algorithms that allows to replace them with other algorithms that could provide various efficiency and power usage. The impact of those algorithms is the research topic of this work. Thus the following hypothesis can be formulated:

For a distributed processing system with reconfiguration ability, does there exist an architecture with algorithms to complete the inputted tasks retaining the operational efficiency and low power consumption?

Contributions

The work done in this dissertation covers all the aspects of the problem stated above in thesis and implements all the goals listed. Summary of the contributions of this work:

- in-depth and comprehensive analysis of the current state of art of the reconfigurable distributed processing systems (Chapter 2)
- formulation of a research problem in the area of distributed processing with reconfiguration capabilities (Chapter 3)
- formulation of evaluation criteria for the stated problem (Chapter 5)
- full and comprehensive design of the distributed architecture satisfying the formulated problem (Chapter 4)
    - the physical architecture of the solution
    - operational algorithms
    - optimization mechanisms
- experimental study of the architecture, algorithms and mechanisms to evaluate their efficiency (Chapter 5)
- creating the experimental platform that implements the proposed architecture and allows implementing new algorithms and additional mechanisms if needed
- formulating future goals (Chapter 6).

CHAPTER 4

RECONFIGURABLE SYSTEM ARCHITECTURE – DPRS

This chapter describes the proposed architecture of *Distributed Processing Reconfigurable System (DPRS)*. It includes the layered architecture, object-oriented approach, interfaces, objects' structure, operational algorithms and few other mechanisms. All these elements have been designed solely for the purpose of this work and constitute the dissertation contributions.

4.1. Proposed notation

In order to clearly describe the system, the following notations are proposed:

Object – is an element of the system that can also have properties.

obj – an object

$\{el\}$ – a set of elements of type *el* (elements denoted as lowercase)

$\{W\}$ – a set denoted as *W* (uppercase), containing multiple elements

$V|\Psi^{v}_{\substack{criterion\ 1\\ criterion\ 2\\ \dots\\ criterion\ n}}\{W\}$ – a selector operator $|\Psi$, that applies the listed criteria to each element *v* from set *V* and yields the elements satisfying the criteria as the new set $\{W\}$

4.2. Overview of the DPRS

Architecture of the proposed approach involves decentralized management. Instead of concentrating on controlling designated nodes (and dividing the roles in the system to management

and non-management), it is proposed to move the management to the nodes to increase flexibility. This way each node can submit the computational task (application), which it manages as in the figure (Fig. 6).

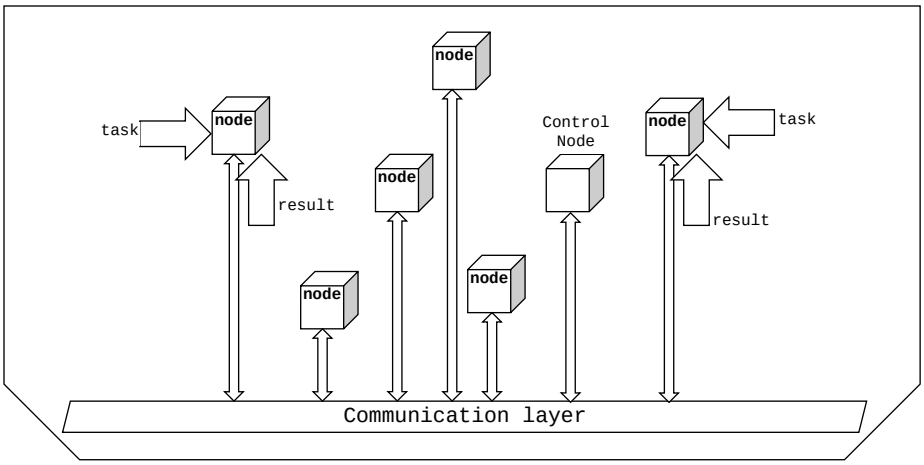


Fig. 6. High level view of the proposed system

This approach also means, that nodes are autonomous – and instead of replying to commands, they operate their own algorithms and request the desired data or elements from other nodes. Such a collaboration of nodes is based on their mutual agreement and interaction, and provides high flexibility to the system operation.

The proposed design includes a practical approach – so no impractical assumptions are made. One example of such an assumption, often found in scientific research regarding distributed processing is that nodes continually know the state of other nodes, or have other information or knowledge about other parts of the system (communication part is simplified/skipped). As the nodes in DPS must coordinate the information, some mechanisms are proposed to provide the inter-node communication. The proposed solution is to introduce a special role, called ROLE_CONTROL. Nodes playing such a role are the ones that coordinate the communication between other nodes, and there might be multiple control nodes present in the system. ROLE_CONTROL nodes

maintain a database with the resources' locations in the system, and each node knows at least one ROLE_CONTROL node. Each node contains one or many processing units that can be CPU, FPGA or other types. As mentioned earlier, each node can introduce its task and get it processed by the distributed system (other nodes can participate in computation). A node introducing the computational task is called *task owner* and bears the role of ROLE_TASK_OWNER.

The ultimate goal of the work is to design a reconfigurable distributed processing system that will possess the following properties:

- distributed processing system with reconfigurable nodes
- system management is decentralized to the maximum possible extent
- nodes are autonomous
- one or more nodes to take the lead as control nodes
    - possibility to define processing functions
- possibility to define tasks to be processed, including data and required functions
- possibility to define function implementation
- getting the output of tasks that are processed by functions
- multiple logical layers – such as processing layer, communications layer.

Also, system must provide a high level of flexibility with multiple points in which the operational algorithms can be defined. Unlike other known systems, the one proposed is designed to be suitable for various applications, and also provide configuration parameters used to tune up the DPS for specific applications.

Layered architecture

The layered approach brings another level of flexibility – for example the communication layer might be implemented in various ways. Therefore, at the stage of hardware implementation, the most suitable protocol/platform/medium can be used. The proposed layered architecture is shown in Fig. 7.

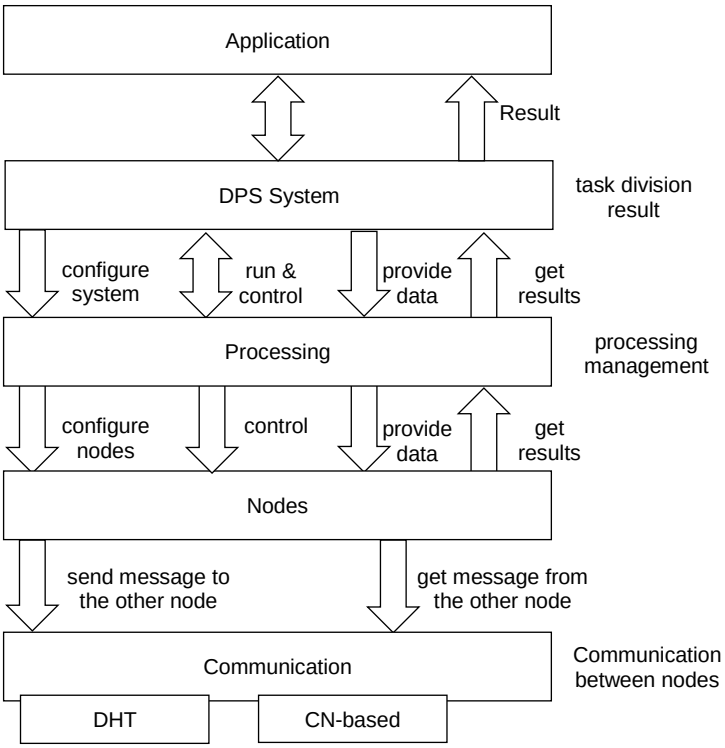


Fig. 7. Layered architecture of proposed solution

Object architecture:

Object oriented modeling is proposed as another architectural property of the proposed design. In such case, most elements of the system are modeled as objects that have some interfacing and properties. The motivation is that real world hardware distributed processing systems usually resemble object-oriented approach, so modeling the DPS using objects brings the advantage of easier implementation of the proposed ideas in practice. Another advantage of the object oriented

approach is the scalable development – nodes (implemented as objects) have the same algorithms deployed and the improvements done in this area mean that all nodes use the updated algorithms / structures so that the system remains compatible as a whole. Nodes are not homogeneous though, as they differ with their local internal properties, such as number and type of processing units, data sources with which they are equipped, efficiency parameters and etc.

4.3. System design and operation

Interfaces

The system architecture is based on the objects – and each object *o* is related to other ones. The relations are implemented through Interface Interconnection Standard (IIS).

Definition

Interface Interconnection Standard (IIS) is the formalization of communication among multiple objects. Each object *o* is equipped with several interfaces $o_{\{a\}}$ included in the interfacing module IM. Each interface has its direction defined:

- Input (I)
- Output (O)
- Bidirectional (B).

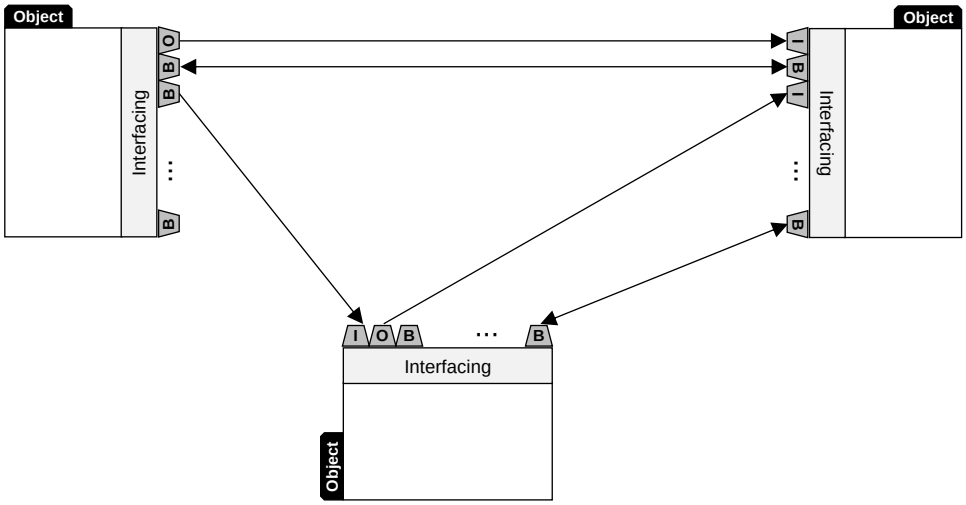


Fig. 8. Structure of IIS

The relations between objects' interfaces are defined in the same way. IIS provides a complete definition of the inter-object communication relations layer and lets designers to separate object-to-object relations from the hardware design (Fig. 8).

Time scale

One of the main properties of the system is that it operates based on finite time periods, called *slots*. The operation time is divided into *T* slots, each slot *t* has its subsequent id. Each slot *t* is considered an *atomic* time period, i.e. only one simple operation can be performed during the time slot *t*.

DPRS structure

This section defines the detailed structure of the components of the DPRS system.

The structure of DPRS is pictured in Fig. 9.

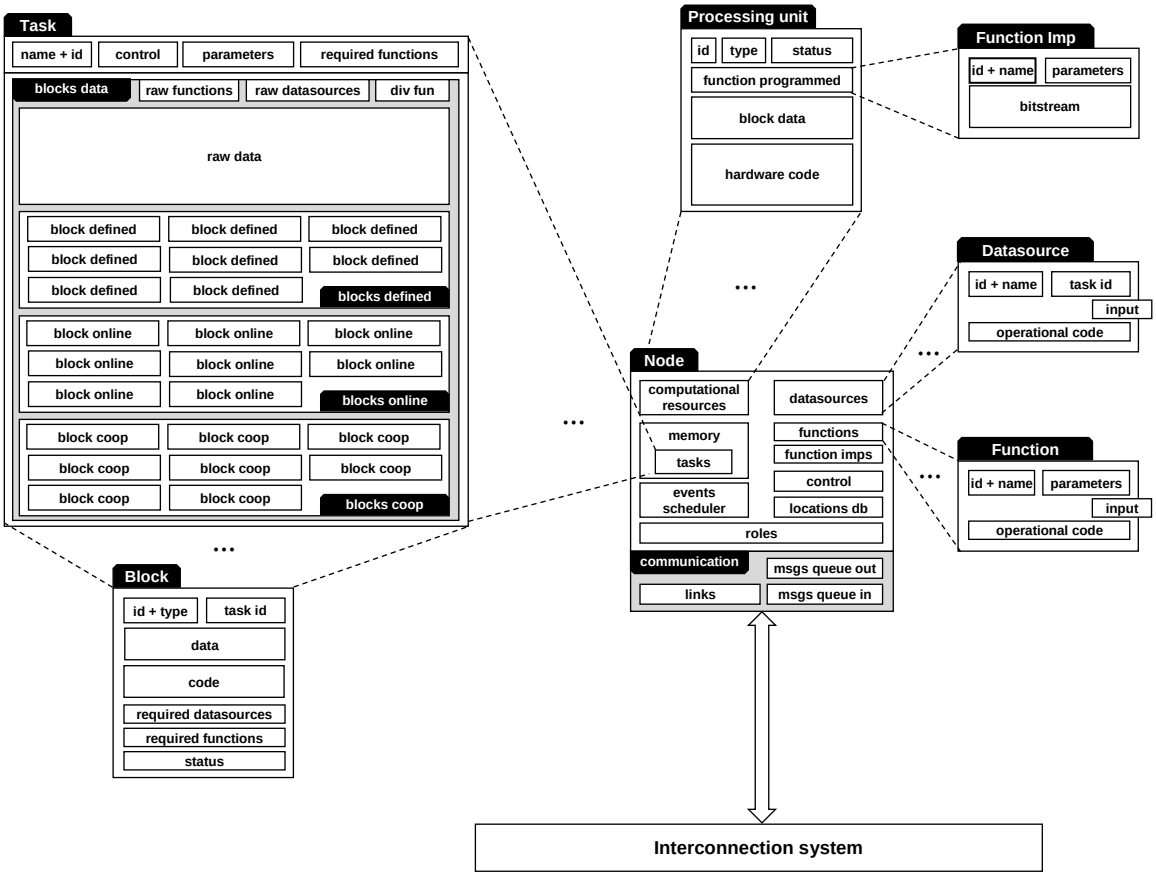


Fig. 9. DPRS architecture

Node is the main component of the proposed distributed processing system. In the proposed DPRS system, each node operates autonomously and makes decisions based on its internal algorithms (unlike traditional approaches, where participating nodes are managed by some root/central units). Due to large number of nodes present in DPS, the proposed algorithms must be efficient, flexible and scalable. As mentioned earlier, the system is built using multiple nodes, with no specialized management element.

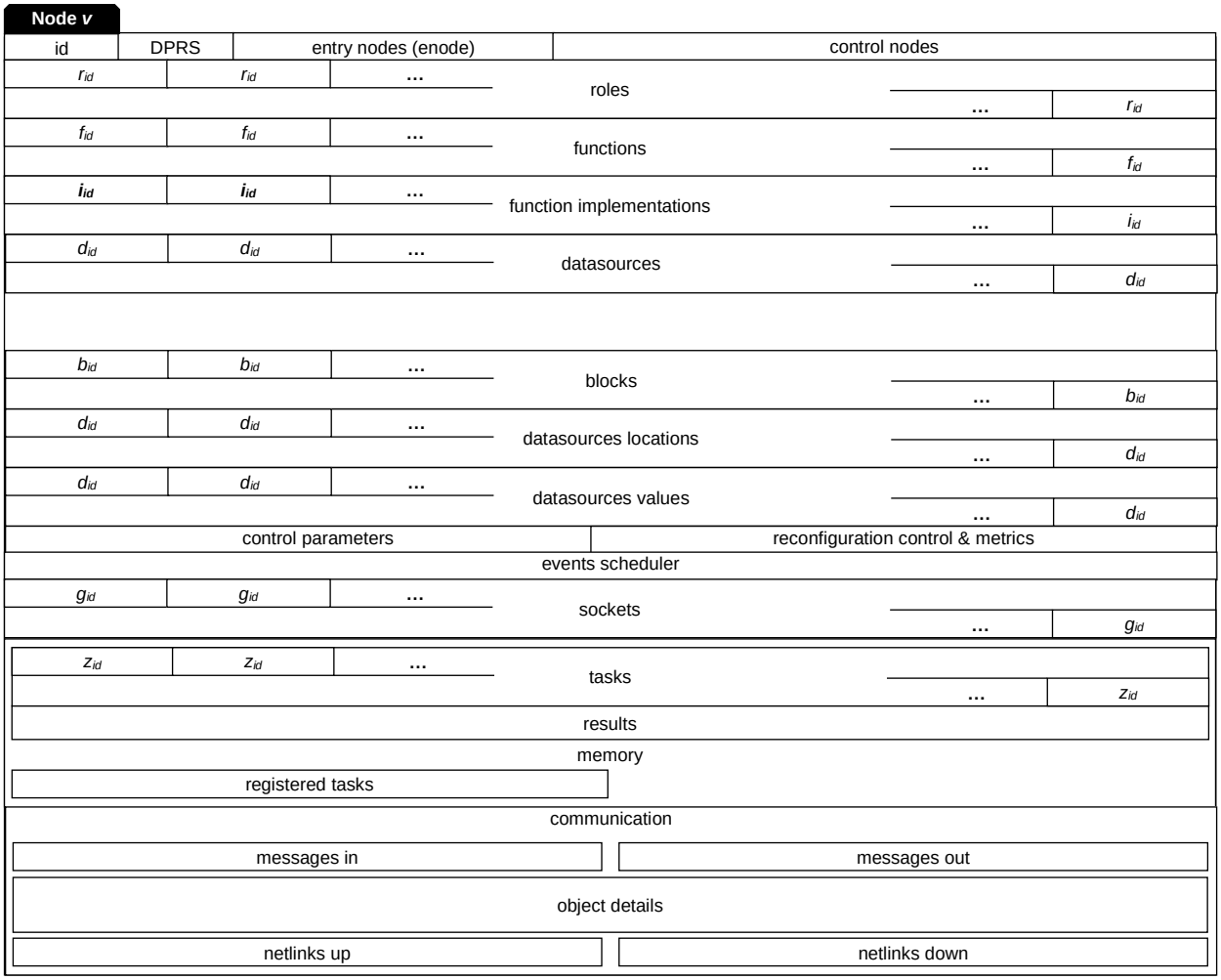


Fig. 10. Architecture of *Node* object

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
        <header>
                <id>#ID</id>
                <DPRS>#DPRS</DPRS>
                <entry>
                        <enode id=1>#EN1</enode>
                        <enode id=2>#EN2</enode>
                        …
                        <enode id=enn>#ENn</enode>
                </entry>
                <control_nodes>
                        <cnode id=1>#CN1</cnode>
                        <cnode id=2>#CN2</cnode>
                        …
                        <cnode id=cnn>#CNn</cnode>
                </control_nodes>
                <roles>
                        <crole id=1>#CR1</crole>
                        <crole id=2>#CR2</crole>
                        …
                        <crole id=crn>#CRn</crole>
                </roles>
                <control_params>
```

```
                <cpar id=1>#CPA1</cpar>
                <cpar id=2>#CPA1</cpar>
                …
                <cpar id=cpan>#CPAn</cpar>
        </control_params>
</header>
<operation>
        <events>
                <evt id=1>#EVT1</evt>
                <evt id=2>#EVT2</evt>
                …
                <evt id=evtn>#EVTn</evt>
        </events>
        <sockets>
                <socket id=1>#SOCKET1</socket>
                <socket id=2>#SOCKET2</socket>
                …
                <socket id=sockn>#SOCKETn</socket>
        </sockets>
</operation>
<memory>
        <local>
                <tasks>
                        <ltask id=1>
                                <task>#TASK</task>
                                <results>#RESULTS</results>
                        </ltask>
                        <ltask id=2>#LTASK2</ltask>
                        …
                        <ltask id=ltn>#LTASKn</ltask>
                </tasks>
        </local>
        <registered_tasks>
                <rtask id=1>#RTASK1</rtask>
                <rtask id=2>#RTASK2</rtask>
                …
                <rtask id=rtn>#RTASKn</rtask>
        </registered_tasks>
        <functions>
                <function id=1>#FUNCTION1</function>
                <function id=2>#FUNCTION2</function>
                …
                <function id=fnn>#FUNCTIONn</function>
        </functions>
        <function_implementations>
                <funimp id=1>#FUNIMP1</funimp>
                <funimp id=2>#FUNIMP2</funimp>
                …
                <funimp id=fin>#FUNIMPn</funimp>
        </function_implementations>
        <blocks>
                <defined>
                        <bdefined id=1>#BD1</bdefined>
                        <bdefined id=2>#BD2</bdefined>
                        …
                        <bdefined id=nbdn>#BDn</bdefined>
                </defined>
                <online>
                        <bonline id=1>#BO1</bonline>
                        <bonline id=2>#BO2</bonline>
                        …
                        <bonline id=nbon>#BOn</bonline>
                </online>
                <cooperative>
                        <bcoop id=1>#BC1</bcoop>
                        <bcoop id=2>#BC2</bcoop>
                        …
                        <bcoop id=nbcn>#BCn</bcoop>
                </cooperative>
                <offline>
                        <boof id=1>#BF1</boof>
```

```
                        <boof id=2>#BF2</boof>
                        …
                        <boof id=nbon>#BFn</boof>
                </offline>
        </blocks>
        <datasources_locations>
                <ds_loc id=1>#DS_LOCATION1</ds_loc>
                <ds_loc id=2>#DS_LOCATION2</ds_loc>
                …
                <ds_loc id=dsln>#DS_LOCATIONn</ds_loc>
        </datasource_locations>
        <datasource_values>
                <ds_val id=1>#DS_VALUE1</ds_val>
                <ds_val id=2>#DS_VALUE2</ds_val>
                …
                <ds_val id=dsvn>#DS_VALUEn</ds_val>
        </datasource_values>
        <reconf_ctrl_metrics>
                <rc_metric id=1>#RC_METRIC1</rc_metric>
                <rc_metric id=2>#RC_METRIC2</rc_metric>
                …
                <rc_metric id=rcmn>#RC_METRICn</rc_metric>
        </reconf_ctrl_metrics>
</memory>
<communication>
        <messages>
                <inq>
                        <message id=1>#MESSAGE1</message>
                        <message id=2>#MESSAGE2</message>
                        …
                        <message id=inqmn>#MESSAGEn</message>
                </inq>
                <outq>
                        <message id=1>#MESSAGE1</message>
                        <message id=2>#MESSAGE2</message>
                        …
                        <message id=outqmn>#MESSAGEn</message>
                </outq>
        </messages>
        <netlinks>
                <up>#NETLINKS_UP</up>
                <dn>#NETLINKS_DN</dn>
        </netlinks>
        <objdata>
                <object id=1>#OBJDATA1</object>
                <object id=2>#OBJDATA2</object>
                …
                <object id=objdn>#OBJDATAn</object>
        </objdata>
</communication>
</node>
```

Each node is assigned to some DPRS system, whose id is stored at the node (multiple instances of DPRS are allowed over the same set of nodes). The structure of the node is shown in Fig. 10, along with its XML definition. The node maintains a list of addresses of entry nodes – this list starts with one or more entry and is updated during the system operation (Fig. 11). Such a list is required for a node joining the system, to be able to find the entry point to the network. DPRS system is decentralized, however for the sake of optimization, nodes with special roles/duties must be

present in the system. Therefore, the roles mechanism is introduced. Indicator $q_{v,r}$ = 1 if role r is assigned to node v. Each node v holds one or more roles ($v_{\{r\}}$): $\sum_{r=1}^{R} q_{v,r} > 0$. Number and characteristics of roles might be defined for each system separately, however it must include basic set of roles that is listed in Tab. 1.

Table 1. Basic roles in the DPRS system

Role	Description
ROLE_PROCESSING	node requests the blocks for the processing and is able to yield the result
ROLE_CONTROL	node performs the management functions: register tasks and respond to processing nodes requests.
ROLE_TASK_OWNER	node has the task in its memory and responds to the block requests, also collecting the results from *processing* nodes
ROLE_BASIC	the default role, to perform the initial tasks such as the service discovery
ROLE_RECONFIGURABLE	role automatically assigned to processing unit with reconfigurable capabilities

Each node maintains the list of *control* nodes, this list is updated during system operation – as nodes can be assigned with roles dynamically. A set of the functions F_z is required to process blocks that are included in task *z*. Each function *f* has an *id* assigned and its software code must be present (Fig. 12). Node stores function software codes and uses them for block processing. The codes can be programmed into the node before system operation, or they can be downloaded from other nodes if required. In order to process a block, the node will have to get the function software code, if that is not present in the node already.

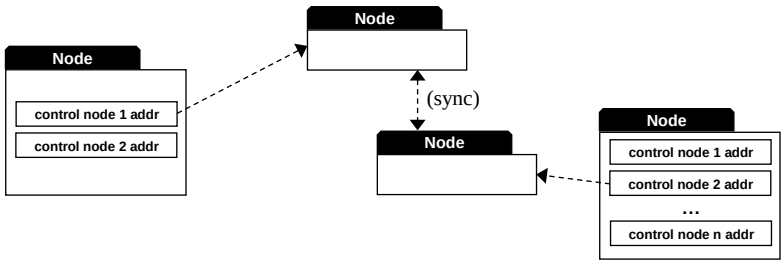


Fig. 11. Control nodes

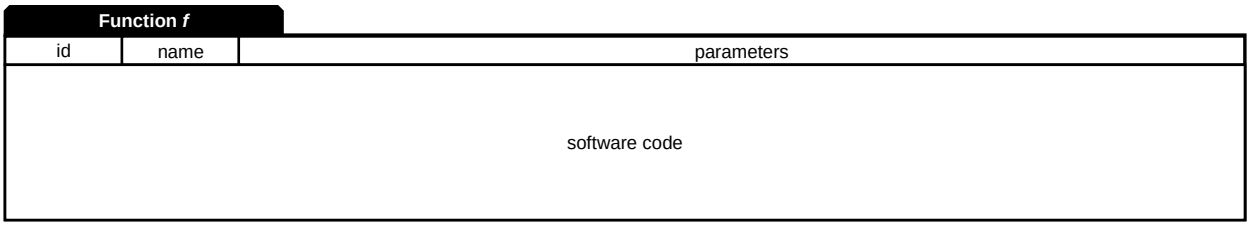


Fig. 12. Architecture of *Function* object

```
<?xml version="1.0" encoding="UTF-8"?>
<function>
        <header>
                <id>#ID</id>
                <name>#NAME</name>
                <parameters>
                        <par id=1>#PAR1</par>
                        <par id=2>#PAR2</par>
                        …
                        <par id=parn>#PARn</par>
                </parameters>
        </header>
        <software_code>
                #softcode
        </software_code>
</function>
```

Function implementations $i = 1, 2, \ldots, F$ are the bitstreams that can be programmed into FPGA type processing units (Fig. 13). Decisions about downloading a bitstream to program local FPGA units are decided by each node autonomously using AL_RECONFIGURE_FPGA algorithm.

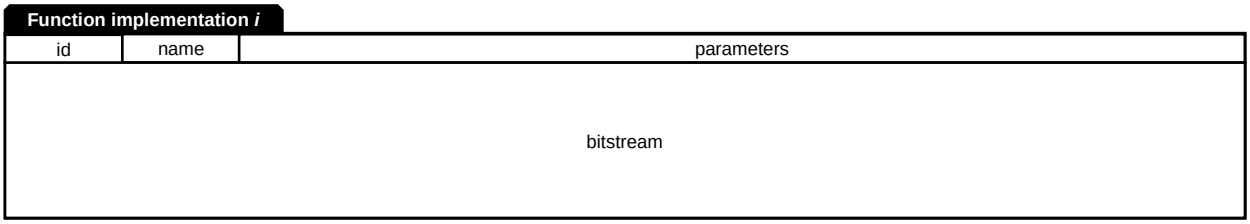


Fig. 13. Architecture of the *Function Implementation* object

```
<?xml version="1.0" encoding="UTF-8"?>
<function_implementation>
        <header>
                <id>#ID</id>
                <name>#NAME</name>
                <parameters>
                        <par id=1>#PAR1</par>
                        <par id=2>#PAR2</par>
                        …
```

```
                <par id=parn>#PARn</par>
            </parameters>
        </header>
        <bitstream>
            #bitstreamraw
        </bitstream>
</function_implementation>
```

Each node *v* is equipped with zero or more datasources *d*: $v_{\{d\}}$, indicator $k_{v,d} = 1$ for each datasource *d* present on node *v*, $\sum_{d=1}^{D} k_{v,d} \geq 0$. *Datasource* is electronic device capable of gathering some specific data from the environment and convert them into digital values, e.g. sensors, cameras, detectors – any kind of devices that can supply some certain kind of data (Fig. 14).

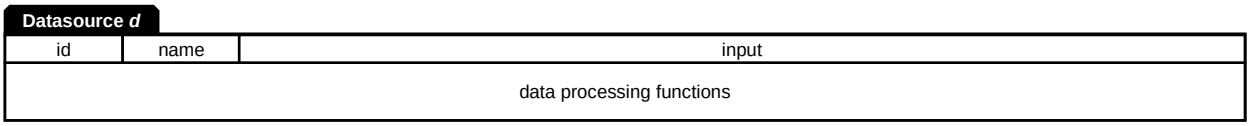


Fig. 14. Architecture of the *datasource* object

```
<?xml version="1.0" encoding="UTF-8"?>
<datasource>
        <header>
            <id>#ID</id>
            <name>#NAME</name>
            <input>
                <input id=1>#INPUT1</input>
                <input id=2>#INPUT2</input>
                …
                <input id=inpn>#INPUTn</input>
            </input>
        </header>
        <data_processing_functions>
            <dpf id=1>#DPF1</dpf>
            <dpf id=2>#DPF2</dpf>
            …
            <dpf id=dpfn>#DPFn</dpf>
        </data_processing_functions>
</datasource>
```

During the system operation, nodes fetch datasources values from many distant nodes, while multiple blocks can require the same datasources. Each node maintains the list of datasources locations (locations are obtained from the *control* node) along with the most recent value for each

datasource. Each datasource value object has an expiration value set that states how long such value is valid and suitable to use.

Nodes are equipped with the memory, where tasks maintained by a node are stored. Memory is also used to store the processing results. Certain node *v* stores only those tasks and results, for which *v* is the task owner. Nodes with the ROLE_PROCESSING use the memory to store the information about tasks registered in the system and for blocks that are currently processed. Each ROLE_TASK_OWNER node manages at least one task of the structure shown in Fig. 20. Task *z* is defined as the computational problem, which needs computation in order to be solved. Task is considered to be an application that can be executed over DPRS. Hereby work proposes the new idea of task definition in order to provide the flexibility of implementing various applications in the distributed manner. The most important elements of the task are the blocks (Fig. 15).

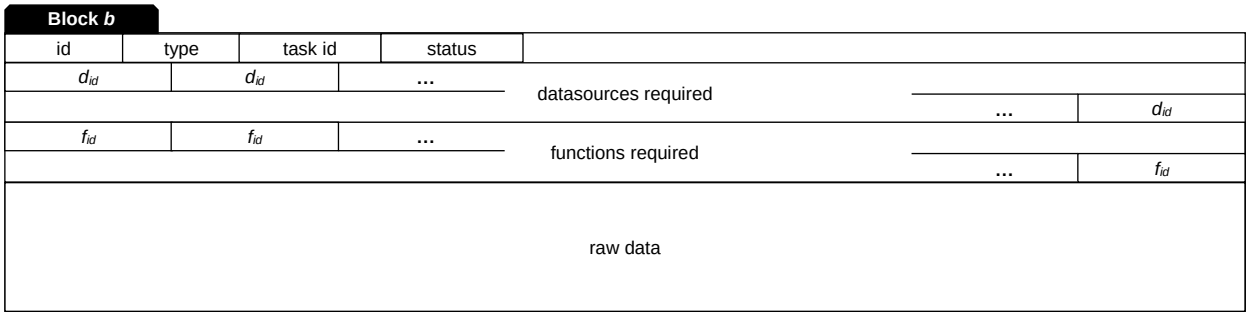


Fig. 15. Architecture of the *block* object

```
<?xml version="1.0" encoding="UTF-8"?>
<block>
        <header>
                <id>#ID</id>
                <type>#TYPE</type>
                <task_id>#TASKID</task_id>
                <status>#STATUS</status>
        </header>
        <ds_required>
                <dsr id=1>#DSR1</dsr>
                <dsr id=2>#DSR2</dsr>
                …
                <dsr id=dsrn>#DSRn</dsr>
        </ds_required>
        <fn_required>
```

```
            <fnr id=1>#FNR1</fnr>
            <fnr id=2>#FNR2</fnr>
            …
            <fnr id=fnrn>#FNRn</fnr>
        </fn_required>
        <rawdata>
            #rawblockdata
        </rawdata>
</block>
```

Blocks are processed on processing units (on nodes). By processing it is meant computing or other type of the process that is done over data embedded in the block or other defined action.

Blocks might be one of four types, for task *z*:

- *blocks defined* – set of blocks that are defined at the time of defining a task. Require one-time processing and the result of processing is available for other blocks. B_{z1} indicates the number of defined blocks for task *z*.
- *blocks online* – set of blocks that are defined at the time of defining a task. They require to be constantly processed through the whole time that a task is being processed in the system. Must be processed by a node that locally has all the data sources required by the block. B_{z2} indicates the number of online blocks for task *z*.
- *blocks cooperative* – set of blocks that are defined at the time of defining a task. They are constantly processed, but their processing may finish earlier than the processing of the whole task. They can request any datasource in the system (both local and remote). B_{z3} indicates the number of cooperative blocks for task *z*.
- *blocks offline* – are not part of task definition. *Blocks offline* is a set of blocks that are created by *divideFunction* function and require one-time processing, after which the *task owner* receives the result. Each block from this set requires all functions from set $\{f_z\}$ and all datasources from set $\{d_z\}$. These blocks are stored at the task owner node. B_{z4} indicates the number of offline blocks for task *z*.

The *task id* property determines which task the block belongs to ($x_{b,z}$ = 1 if block *b* belongs to task *z*). Block *b* processed on node *v* can be assigned one of the following statuses:

- BL_STATUS_IDLE – no actions were taken on the block so far
- BL_STATUS_SENT – block was sent for computation from the *task owner* node to the *processing* node *v*
- BL_STATUS_PROCESSING – the processing of the block started on node *v*
- BL_DS_LOCATIONS_WAITING – node *v* is waiting for the locations of datasources required for processing block *b*
- BL_DS_KNOWN – node *v* obtained all the locations of the datasources $b_{\{d\}}$, that are required to process block *b*
- BL_WAITING_FOR_DS – node *v* is waiting for the values of datasources from distant node(s)
- BL_STATUS_READY – node *v* received all distant datasource values, thus all the datasource values are available on the node *v*
- BL_STATUS_PROCESSED – block *b* has finished processing

The workflow for the type of block defined and/or offline is shown in Fig. 16. Once the processing is started, it needs to be determined if the process requires any datasources. If it does, then it is checked if the datasources are all available locally (i.e. relevant hardware is installed on the same node). If there is a need for fetching values from remote datasources, node needs to determine if the location(s) of these remote datasource(s) are known – and if not, if there is a need to obtain their locations. If yes, the locations are obtained and then the request for the datasource values is sent. Once all the datasource values are ready (remote and/or local) the processing starts, and once

it is done, the result is sent out to the ROLE_TASK_OWNER node, and block is considered as finished. The right part of Fig. 16 shows the status of a block at each stage of the above process.

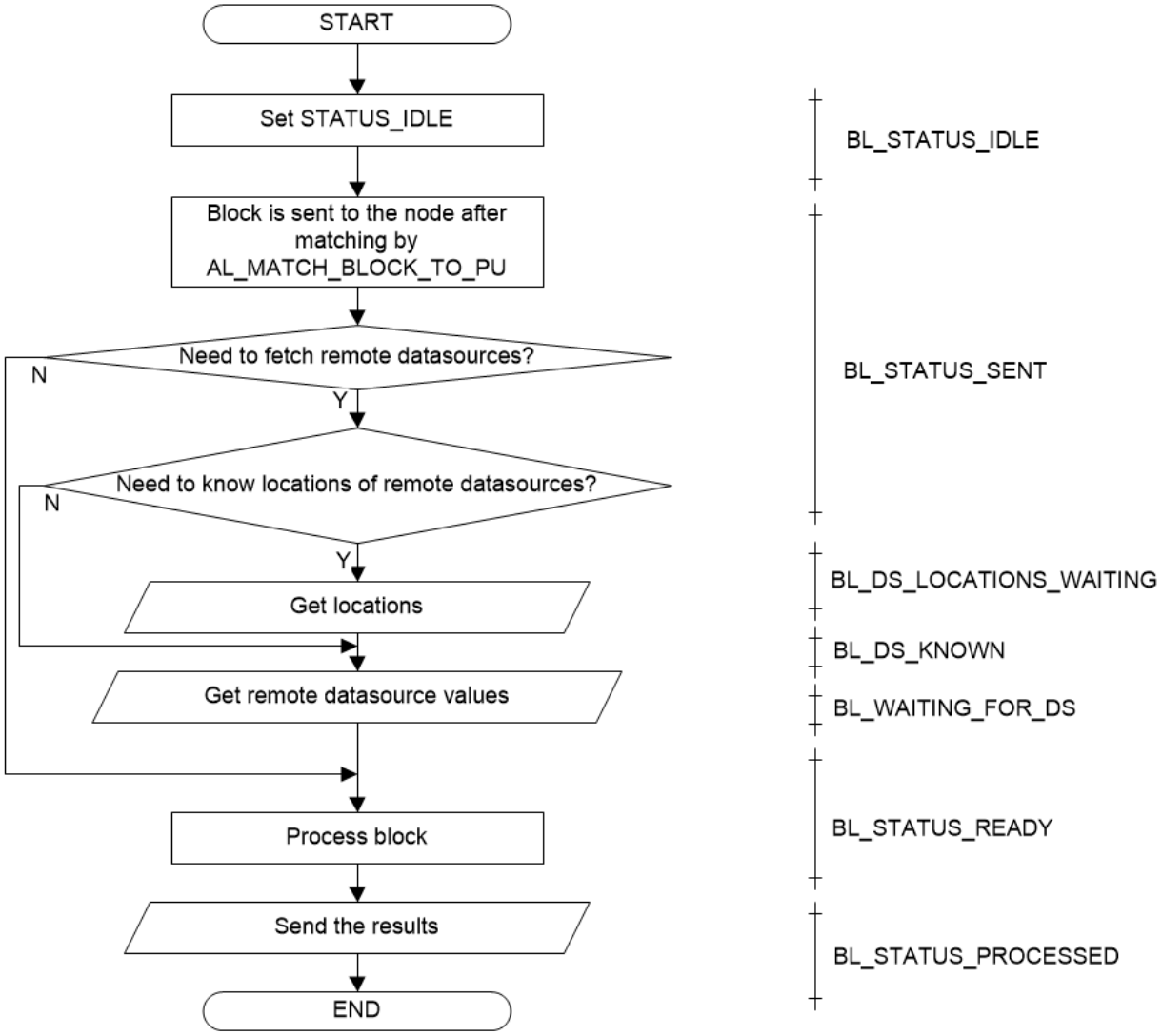


Fig. 16. Workflow for blocks defined and blocks offline

Blocks online and blocks cooperative can have two additional statuses:

- BL_STATUS_ONLINE_PROCESSING – block's online processing is in progress
- BL_STATUS_STOPPED – block's online processing has been stopped.

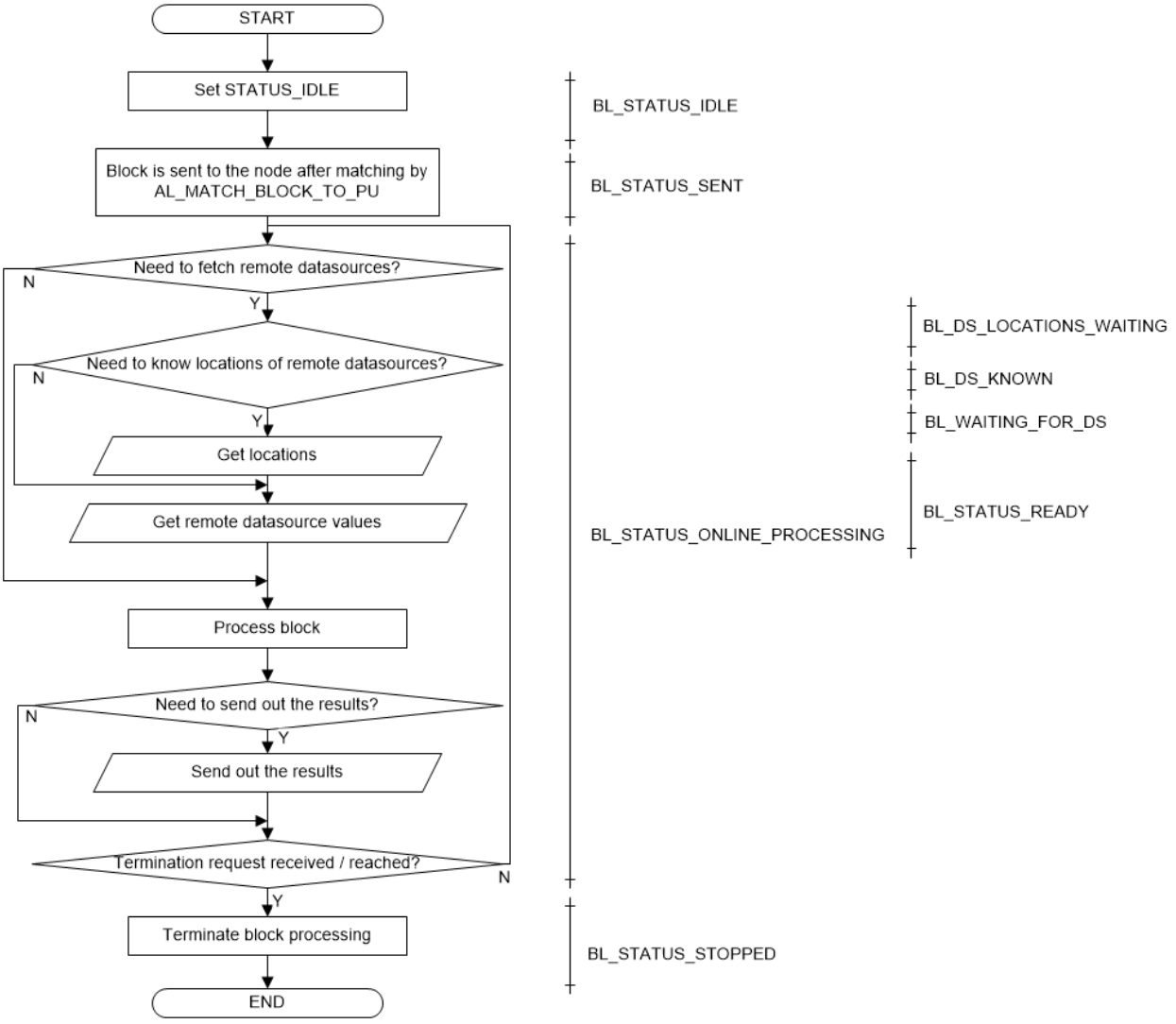


Fig. 17. Workflow for block cooperative

Blocks cooperative and blocks online have different workflow (Fig. 17 and Fig. 18). For blocks cooperative, the processing is done in the loop that may also include waiting on requests for data. Thus, cooperative block can stay active and be constantly processed during the entire time the task/application is active. Blocks online work in a very similar way. There are two major differences between blocks cooperative and blocks online: 1) blocks online can operate only on local datasources, while blocks cooperative can use local datasources, and also request values of

remote datasources 2) blocks cooperative can be terminated anytime, while blocks online stay active for the entire lifetime of the task/application.

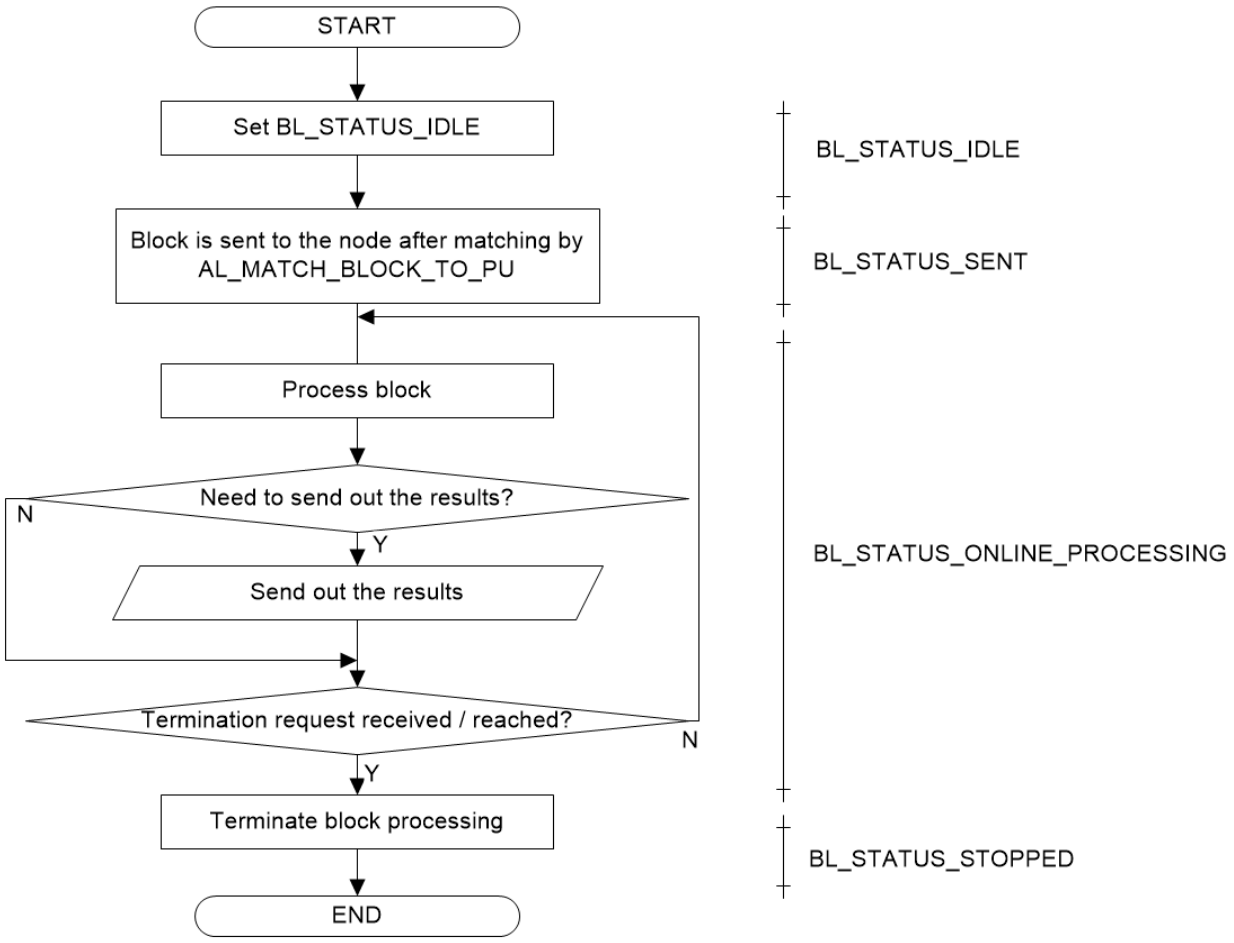


Fig. 18. Workflow for block online

Block *b*, according to its definition, may require zero or more datasource values in order to be processed ($\sum_{d=1}^{D} l_{b,d} \geq 0$), $l_{b,d}$ indicates that block *b* requires datasource *d.* If such values are not available on the node *v* processing the block, then node *v* requests the values of $\{d\}$ from the node(s) that are equipped with required datasources. DPRS implements restrictions for the

locations of datasources: the task designer is able to specify, whether non-local datasource values are allowed:

- b_L = 1: if all of the datasource values required by block *b* must be present on the node *v* in order to perform block processing on node *v*;
- b_L = 0: values of datasources absent on node *v*, but required by block *b*, can be requested and obtained by node *v* from the other nodes that are equipped with these datasources.

Property b_L is set to 1 for all *online* blocks by default. Each block requires one or more function to be processed: ($\sum_{f=1}^{F} n_{b,f} > 0$), for $n_{b,f}$ = 1 when function *f* is required by block *b*. The list of ids of required functions is stored in the definition of each block. Raw data is any kind of data that is used for the block processing, such as binary data, look up tables, constants, definitions and etc. It can be empty though, if functions processing the blocks do not require any data and operate just by themselves.

Task has the following properties:

- task has an owner (a node introducing the task)
- task requires one or more functions to be processed
    - required functions are stated in task definition
    - each block contains the information, which functions are required for processing (these functions are the subset of the functions that are enclosed in task)
    - block can request information from distant nodes (their datasources). All the blocks can require data sources – if they are not available locally, then they must be fetched from another node.

The block processing yields a *result* object (Fig. 19):

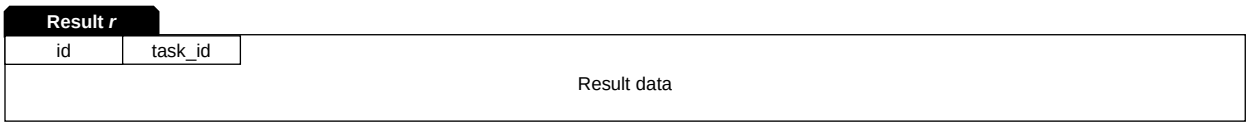


Fig. 19. Architecture of *result* object

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
        <header>
                <id>#ID</id>
                <task_id>#TASKID</task_id>
        </header>
        <resdata>
                #resultdata
        </resdata>
</result>
```

For block of types BLOCK_DEFINED and BLOCK_OFFLINE, the *id* assigned is the same as the block that was a source for processing. For BLOCK_ONLINE and BLOCK_COOPERATIVE the id assigned ranges outside the id:s already used for blocks. This mechanism allows connecting the results with the source data. Each result has task id assigned (to identify to which task the result belongs), and contains a *result data* field that is filled with the results of the block processing.

In this work, an universal task definition is proposed. This definition allows to run various types of applications in the DPRS system requiring various relations between subtasks, different types of processing etc. The structure of the task object is shown in Fig. 20.

B_{z1} – number of *blocks defined* for task z can be expressed as follows:

$$B_{z1} = \sum_{b=1}^{B} b_{t1}\, x_{b,z}$$

where $b_{t1} = 1$ if block b is a type of *block defined*. Similarly, for B_{z2}, B_{z3} and B_{z4}:

$$B_{z2} = \sum_{b=1}^{B} b_{t2}\, x_{b,z} \qquad B_{z3} = \sum_{b=1}^{B} b_{t3}\, x_{b,z} \qquad B_{z4} = \sum_{b=1}^{B} b_{t4}\, x_{b,z}$$

Number of *blocks offline* B_{z4} is determined by the *divideFunction* function. This function runs the process of defining blocks offline using raw data and $\{f_{id}\}$ and $\{d_{id}\}$ elements of the task definition structure.

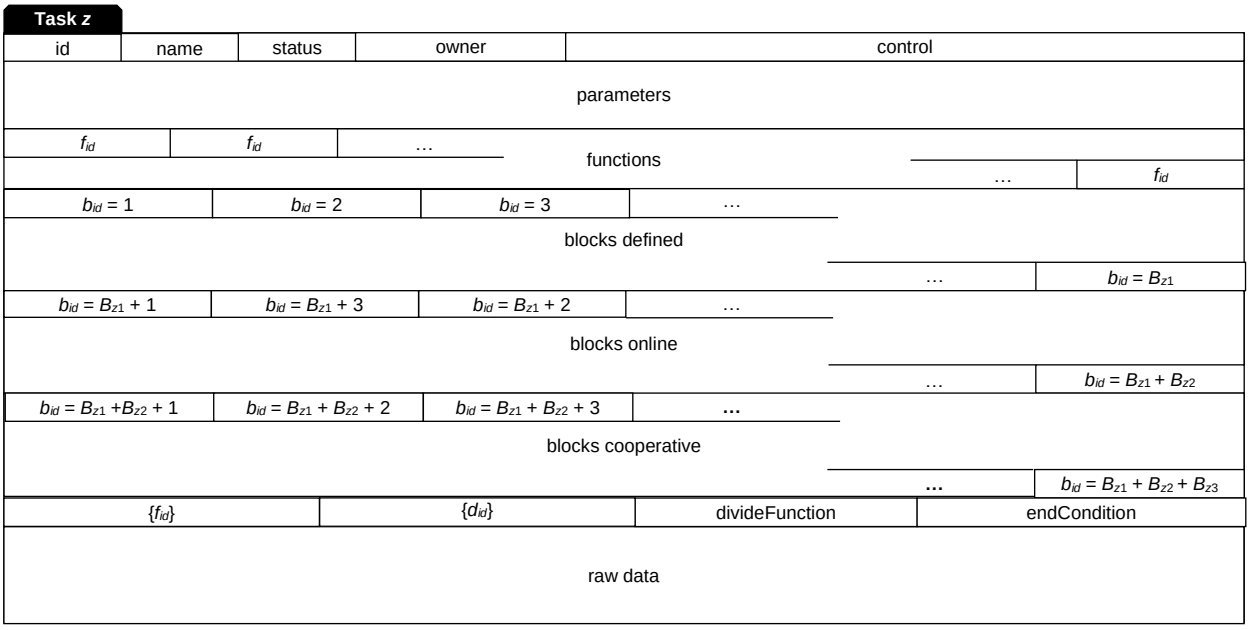


Fig. 20. Architecture of *Task* object

```
<?xml version="1.0" encoding="UTF-8"?>
<task>
        <header>
                <id>#VALUE</id>
                <name>#NAME</name>
                <status>#STATUS</status>
                <owner>#OWNER</owner>
                <control>
                        <control_property id=1>#CP1</control_property>
                        <control_property id=2>#CP2</control_property>
                        …
                        <control_property id=cpn>#CPn</control_property>
                </control>
                <parameter>
                        <parameter_item id=1>#PI1</ parameter_item>
                        <parameter_item id=2>#PI2</ parameter_item>
                        …
                        <parameter_item id=pin>#PIn</ parameter_item>
                </parameter>
        </header>
        <functions>
                <headers>
                        <fheader id=1>#FH1</fheader>
                        <fheader id=2>#FH2</fheader>
                        …
                        <fheader id=fn>#FHn</fheader>
                </headers>
                <implementations>
```

```
                <fimp id=1>#FI1</fimp>
                <fimp id=2>#FI2</fimp>
                …
                <fimp id=fn>#FIn</fimp>
            </implementations>
        </functions>
        <blocks>
            <defined>
                <bdefined id=1>#BD1</bdefined>
                <bdefined id=2>#BD2</bdefined>
                …
                <bdefined id=bdn>#BDn</bdefined>
            </defined>
            <online>
                <bonline id=1>#BO1</bonline>
                <bonline id=2>#BO2</bonline>
                …
                <bonline id=bon>#BOn</bonline>
            </online>
            <cooperative>
                <bcoop id=1>#BC1</bcoop>
                <bcoop id=2>#BC2</bcoop>
                …
                <bcoop id=bcn>#BCn</bcoop>
            </cooperative>
            <offline>
            </offline>
            <blockdata>
                <functions>
                    <function id=1>#FBO1</function>
                    <function id=2>#FBO2</function>
                    …
                    <function id=fbon>#FBOn</function>
                </functions>
                <datasources>
                    <datasource id=1>#DSO1</datasource>
                    <datasource id=2>#DSO2</datasource>
                    …
                    <datasource id=dson>#DSOn</datasource>
                </datasources>
                <divFun>#DIVFUN</divFun>
            </blockdata>
        </blocks>
        <data>
            <raw>
                #RAWDATA
            </raw>
        </data>
</task>
```

Task operation and properties are modeled using the following elements:

- *id* – identifier of task
- *name* – arbitrary name of task, for easy identification
- *status* – current status of task:
    - TASK_INACTIVE – task has not started processing
    - TASK_REGISTERED – task is registered at the task manager node

    - TASK_DIVIDED – all *blocks offline* are created on the *task owner* node
    - TASK_UNREGISTERED – task unregistered at the *task manager* node (but block online might still be running)
    - TASK_DEACTIVATED – processing of all blocks is finished and all the results are collected.
- *owner* – id of the *task owner* node
- *control* – variables, constants and other data used to control the task
- *parameters* – set of parameters that can be used by blocks
- *functions* – definitions of functions, that are used by blocks for processing
- $\{f_{id}\}$ – set of functions identifiers that are required by *blocks offline*.
- $\{d_{id}\}$ – set of data source identifiers that are required by *blocks offline*.
- *divideFunction* – the function that creates *blocks offline*, using raw data and $\{f_{id}\}$ and $\{d_{id}\}$ sets.
- *endCondition* – specifies what condition(s) must be satisfied in order to terminate a task and consider it as finished. Just the fact that blocks offline and blocks defined are processed does not give enough flexibility, as the processing would have to be limited to the time span of their processing. Also, the blocks online operate constantly therefore the *endCondition* is introduced.
- *raw data* – the structure of binary or text data, that will be divided using task's *divide*() function. The result of such a division will be the set of offline blocks.

Fig. 21 shows the flow of task operation. First, task is registered (at ROLE_CONTROL node), and ROLE_TASK_OWNER divides the raw data if necessary. Then the processing starts, and blocks are sent out to the processing nodes and results are collected. If all blocks offline and defined are

processed, then task is unregistered, but the processing can still continue for already allocated blocks online and cooperative. When task end condition is satisfied, then the termination message is sent to all nodes handling the online and cooperative blocks, and the task is terminated.

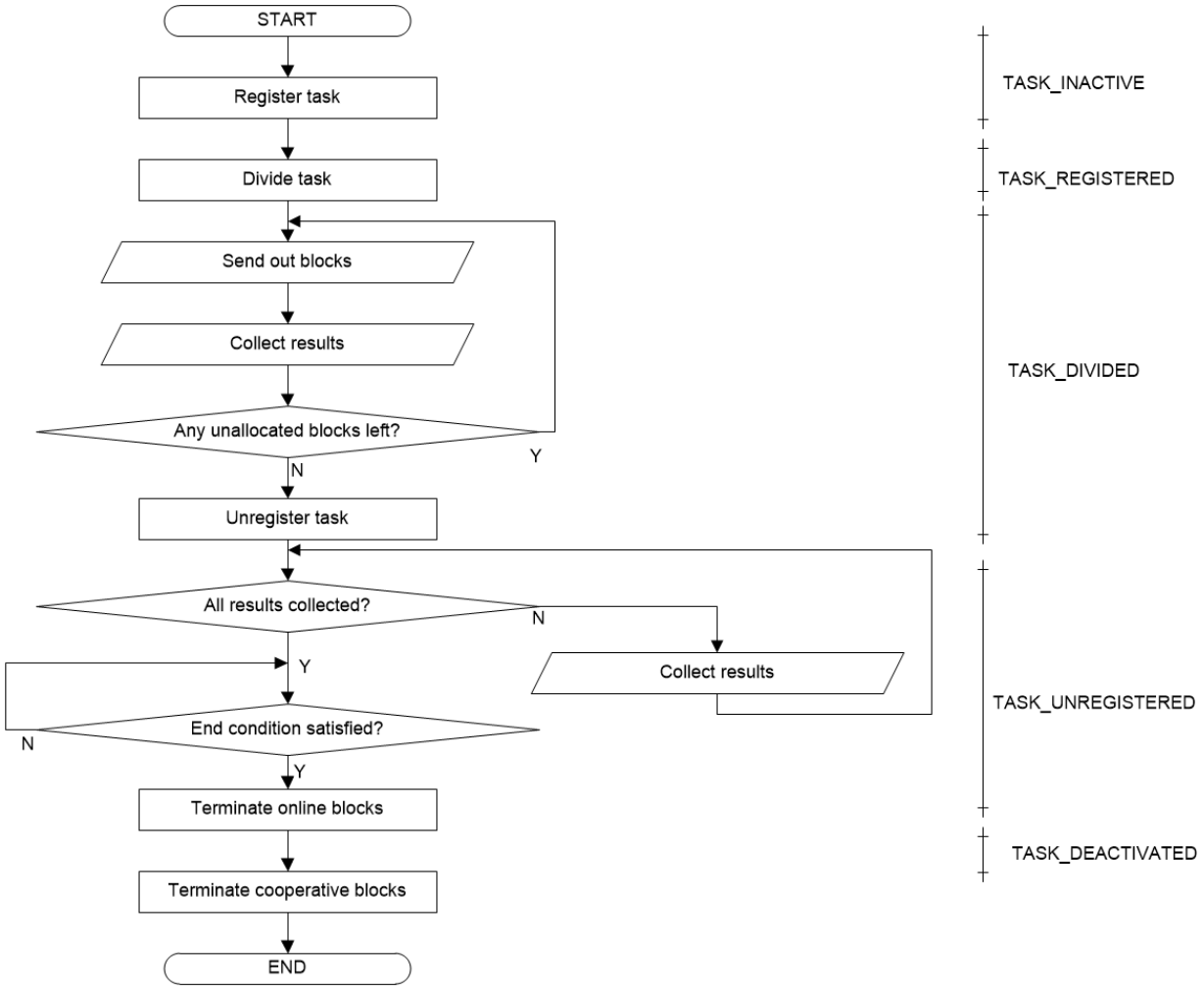


Fig. 21. Task status flow

The processing on nodes is done using processing units. Each node *v* has a given number *G* of sockets *g*: $\sum_{g=1}^{G} w_{v,g}$ ($w_{v,g}$ determines if socket *g* is installed on node *v:* $w_{v,g}$ = 1 if yes, 0 otherwise). Sockets can be filled with *processing units p*. Each node *v* must be equipped at least with one *processing unit*, no more than the number of sockets ($1 \leq \sum_{p=1}^{P} \sum_{g=1}^{G} w_{v,g}\, u_{p,g} \leq \sum_{g=1}^{G} w_{v,g}$). The

$u_{p,g}$ indicator shows if processing unit p is installed in socket g ($u_{p,g}$ = 1; 0 otherwise). The simplified structure of the processing unit object is shown in Fig. 22.

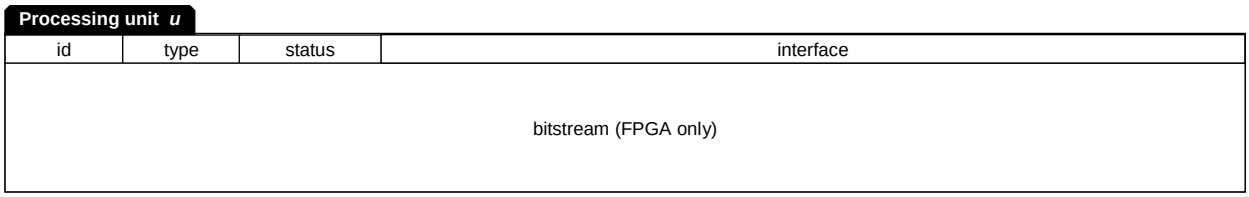


Fig. 22. Architecture of *processing unit* object

```
<?xml version="1.0" encoding="UTF-8"?>
<pu>
        <header>
                <id>#ID</id>
                <type>#TYPE</type>
                <status>#STATUS</status>
        </header>
        <interface>
                <iob id=1>#IOB1</iob>
                <iob id=2>#IOB2</iob>
                …
                <iob id=n>#IOBn</iob>
        </interface>
        <bitstream>
                #bitstreamdata
        </bitstream>
</pu>
```

Each processing unit object is considered as the object to which data, and other values are passed, and the results are returned after processing is done (processing units are also objects using IIS). Processing units have access to the memory, function codes present on the node and other assets present locally. Multiple types of processing units are supported:

- CPU – processing unit with a standard execution environment, hardware is not designed specifically to any of the application
- FPGA – reprogrammable chip that can be programmed for any specific application using bitstreams

- Specialized chips (*ASIC – Application Specific Integrated Circuits*) – specialized chips designed for a specific application. For example DSP, that provides signal processing functions already implemented in hardware.

The scheduling management and blocks assignments are handled by both local and distant algorithms. The operational management is described in section 4.5. Each processing unit *p* can have one of the following states:

- PU_STATUS_IDLE – *p* is not assigned with any processing
- PU_STATUS_PROCESSING – *p* is currently processing a block
- PU_STATUS_PROGRAMMING – *p* is currently programmed with a bitstream (FPGA only)
- PU_STATUS_ALLOCATED – *p* is assigned a block, and is awaiting for the processing.
- PU_STATUS_RECONFIGURATION – running reconfiguration procedures, except programming FPGA (FPGA only)

The operation of non-reconfigurable and reconfigurable units differ. Fig. 23 presents the operation workflow for the former case. The processing unit starts in the PU_STATUS_IDLE mode, and when it receives the block (the request is handled by the node) it obtains the datasources if they are required. Having these values ready, the processing starts and is concluded with the results. Block is deactivated and the processing unit is ready to process next block. For the reconfigurable case, the workflow is the same but includes also the reconfiguration parts (Fig. 24). Each processing unit handles the reconfiguration counter modified each time slot. Once it reaches zero, then the reconfiguration algorithm executes. If the counter reaches zero and the processing still goes on, the reconfiguration algorithm waits till the processing ends. For the reconfiguration procedure, first it is decided if the reconfiguration should be done at that point, and if yes – what

function should be programmed in. In case of the reprogramming, the bitstream containing the decided function is fetched and then programmed into the processing unit's FPGA chip. The process is concluded with reset of the reconfiguration counter that is set to the predetermined value. If the reconfiguration has not been decided, then the setting of the reconfiguration counter is done as well. After that, processing unit continues the blocks processing, if the task is still active.

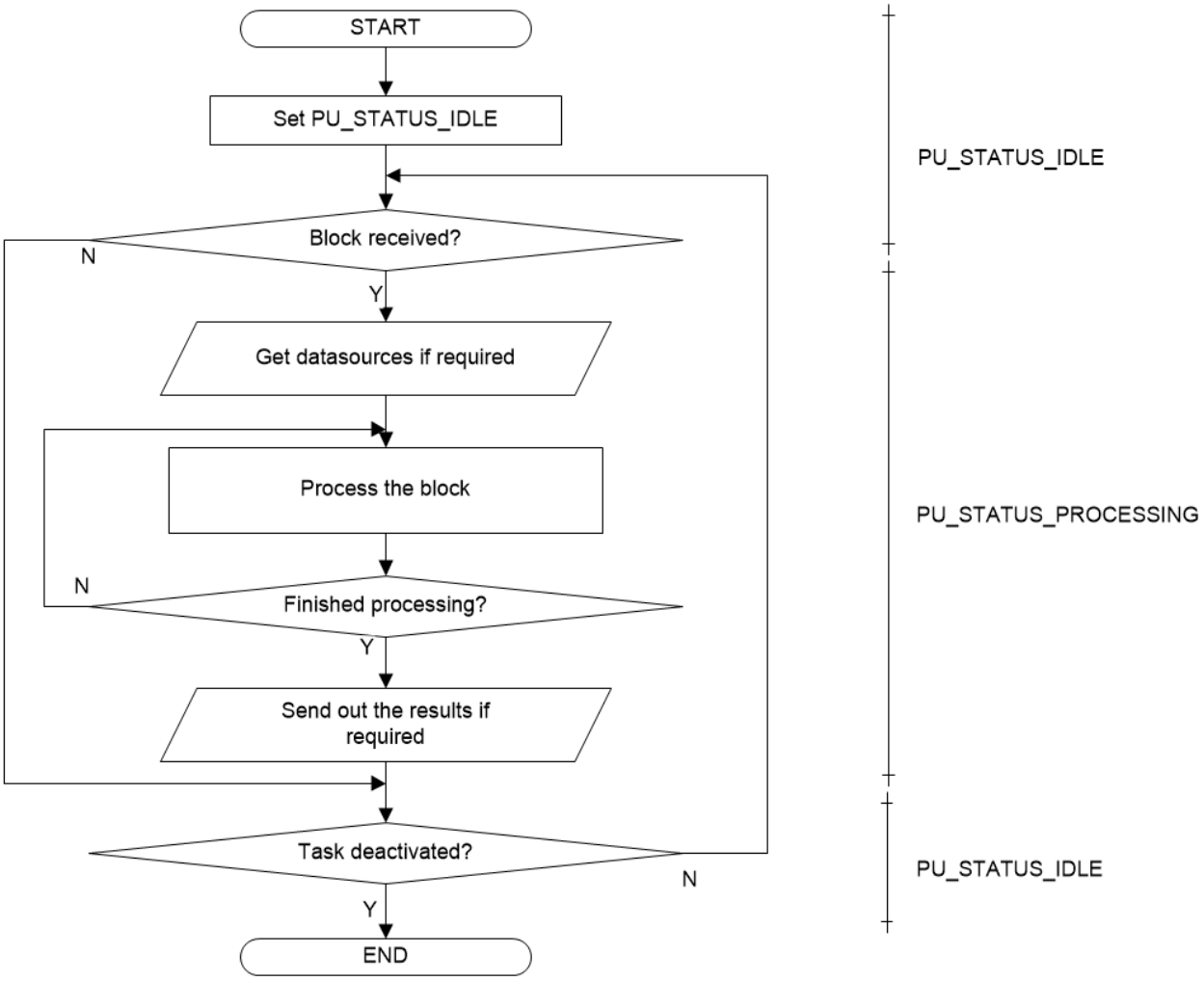


Fig. 23. Workflow for non-reconfigurable processing unit

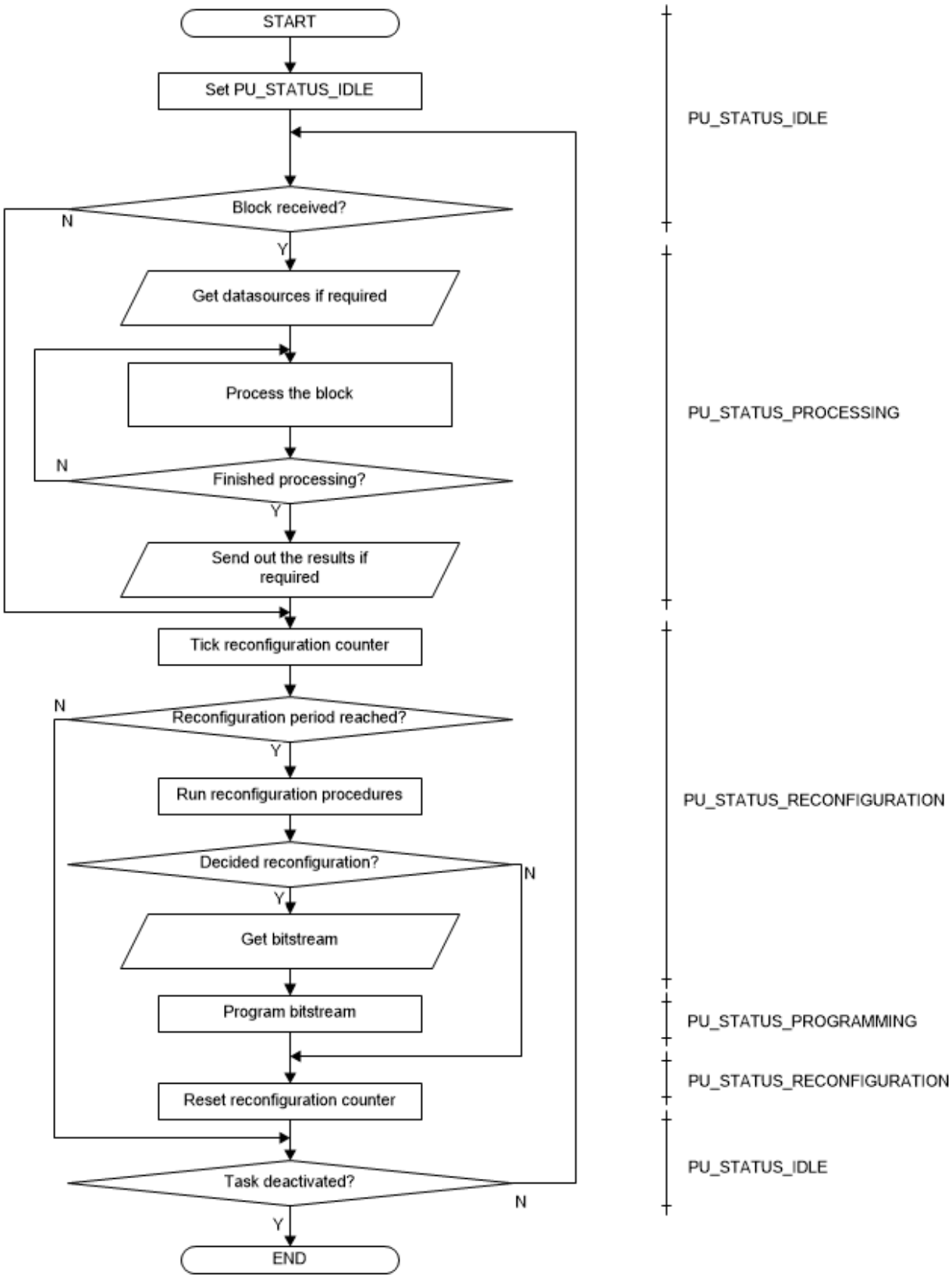


Fig. 24. Workflow for reconfigurable processing unit

DPRS system uses the layered architecture, thus the communication is implemented as a set of objects providing the whole definition of communication system. Each object requiring the communication (e.g. nodes) implements a separate object that is implementing the IIS. Node uses the set of known functions (same set on each node) and these functions must be implemented by the communication layer. This means that having the fully operable system, the communication layer can be exchanged to different implementation, if it's compatible with the one that node uses.

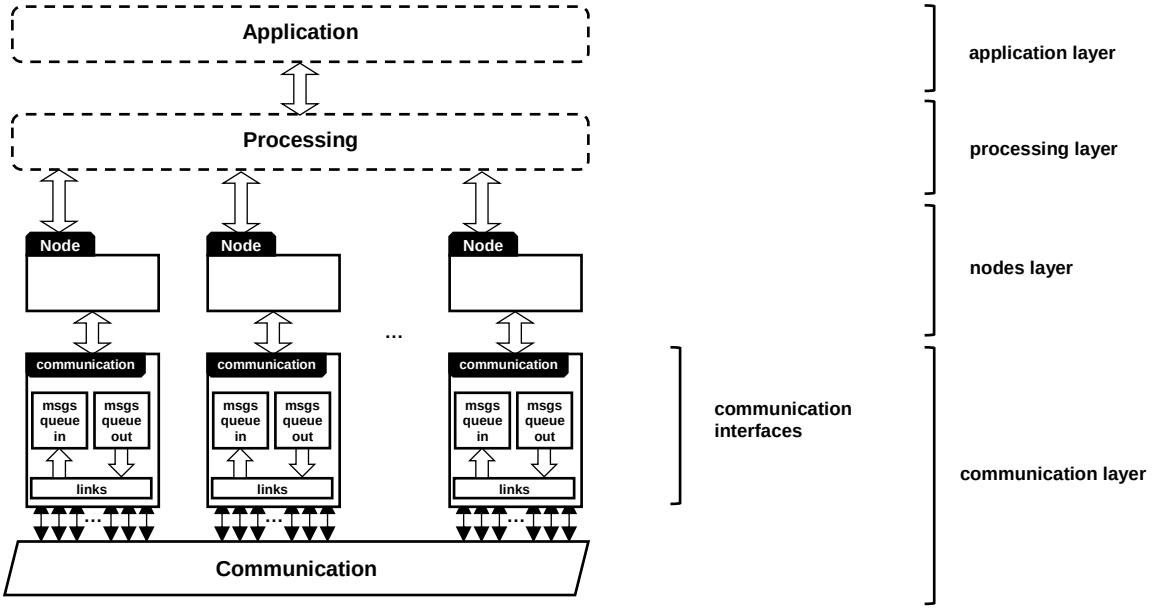


Fig. 25. Idea of communication layer

Each node will implement the common communication object that will be the part of the communication layer (Fig. 25). Each node is characterized by the connection capabilities and upload and download bandwidths (v_{up}, v_{dn}) are shared among all the elements currently transmitting data (1). The transmission speed depends on the speed of source node v and destination node w (that could be also limited due to many blocks transmitted at the time):

$$speed = min\left(\left\lfloor \frac{v_{up}}{\sum_{h=1}^{H} h_v h_{v1}} \right\rfloor, \left\lfloor \frac{w_{dn}}{\sum_{h=1}^{H} h_w h_{w1}} \right\rfloor\right) \quad (1)$$

h_v indicates, that a link h belongs to node v, h_{v1} indicates that link h is used.

Current transmission speeds should be considered when deciding about the bitstream download. The proposed simulator implements all the functions and algorithms required for the proper communication, and the actual data transmission is fully simulated (in terms of speed, time required etc. – details are given in the section 5.2). However the data transmission element will be isolated and ready to replace with some existing solution, e.g. the one using TCP/IP.

The communication system at the logical level is implemented as a set of message types, each with specific properties (Fig. 26).

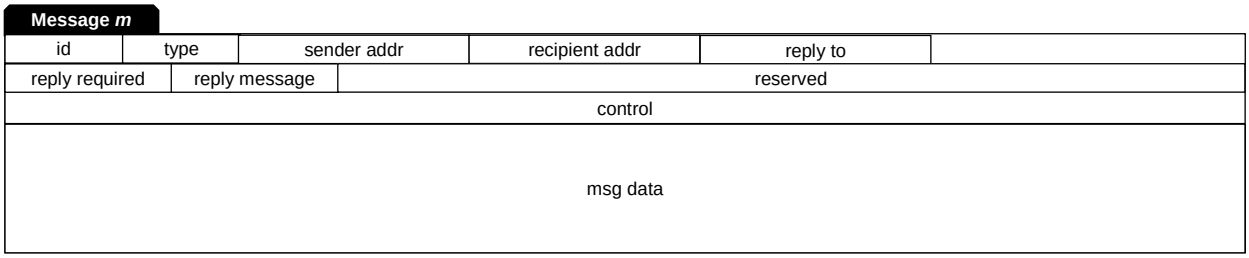


Fig. 26. Architecture of *message* object

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
        <header>
                <id>#ID</id>
                <type>#TYPE</type>
                <sender_addr>#SENDER_ADDR</sender_addr>
                <recipient_addr>#RECIPIENT_ADDR</recipient_addr>
                <reply_to>#REPLY_TO</reply_to>
        </header>
        <settings>
                <reply_required>#RR</reply_required>
                <reply_message>#RM</reply_message>
                <reserved>#RS</reserved>
        </settings>
        <control>
                <cnt id=1>#CNT1</cnt>
                <cnt id=2>#CNT2</cnt>
                …
                <cnt id=n>#CNTn</cnt>
        </control>
```

```
        <reserved>#RS</reserved>
        <msgdata>
                #messagedata
        </msgdata>
</message>
```

The following properties are included in the *message* object:

- *id* – object identificator
- *type* – the type of the message. The following messages are defined:
    - MSGT_REQUEST_CNTNODES – requesting the id:s of control nodes present in the system
    - MSGT_REPLY_CNTNODES – reply to MSGT_REQUEST_CNTNODES message, contains the list of control nodes know to the node issuing MSGT_REPLY_CNTNODES message
    - MSGT_REGISTER_TASK – issued by a node that holds *task owner* role and wants to register a new task for processing. MSGT_REGISTER_TASK can be sent only to the node holding *control node* role.
    - MSGT_REQUEST_TASK_DATA – sent to CN in order to obtain the data about registered tasks present in the system, along with node id:s of their task owners.
    - MSGT_REPLY_TASK_DATA – reply to the message of type MSGT_REQUEST_TASK_DATA, contains the set of pairs of (node id, task id).
    - MSGT_REQUEST_BLOCK – sent from the node bearing ROLE_PROCESSING to the node with role ROLE_TASK_OWNER to request the block for the processing. For the purpose of optimization, requesting node attaches brief information regarding its assets, called *nodeAssets* (Fig. 27):
    - MSGT_REPLY_BLOCK – reply to the MSGT_REQUEST_BLOCK, contains the *block* object enclosed in the *data* section

- MSGT_FUNCTIONS_ATT – sent from a node ROLE_TASK_OWNER to a node requesting a block. The decision about sending this message is determined based on *nodeAssets* object, ROLE_TASK_OWNER node sends the functions that are missing on the node requesting a block, but are missing on this node.
- MSGT_DEACTIVATE_TASK – sent from a node ROLE_TASK_OWNER to nodes requesting blocks, when there are no more blocks to be sent for processing. Node with a role ROLE_PROCESSING stops sending MSGT_REQUEST_BLOCK for task id enclosed in the message MSGT_DEACTIVATE_TASK. Task with this id is removed from *registered tasks* list.
- MSGT_REQUEST_DS_LOCATIONS – message used to get the id:s of nodes, that are equipped with datasources whose id:s are listed in this message. This message is sent to node ROLE_CONTROL from node ROLE_PROCESSING. The information about datasources locations is collected along with other communication: using *nodeAssets* and task registering process.
- MSGT_REPLY_DS_LOCATIONS – sent from node ROLE_CONTROL to node ROLE_PROCESSING. Contains the list of known locations of datasources.
- MSGT_REQUEST_DS_VALUE – message can be sent between nodes of any role. It is issued by a node that needs the current data from a datasource of given id.
- MSGT_REPLY_DS_VALUE – message sent from a node equipped with a certain datasource to another node that requested the datasource value.

- MSGT_RESULT – message sent from a node ROLE_PROCESSING to the ROLE_TASK_OWNER. It contains the data being the result of the block processing.
- MSGT_REQUEST_RECONF_METRICS – control message that is sent from node ROLE_PROCESSING to node ROLE_CONTROL with the inquiry about current task and system state. This data is used at the ROLE_PROCESSING node to make decisions about reconfiguration. MSGT_REQUEST_RECONF_METRICS is always issued before first reconfiguration. Later, it is sent according to the decision of algorithm AL_RECONFIGURE_FPGA.
- MSGT_RECONF_STATUS – sent from ROLE_PROCESSING with ROLE_RECONFIGURABLE to ROLE_CONTROL with the confirmation which function id was selected for programming.
- MSGT_REPLY_RECONF_METRICS – message is sent as a reply to MSGT_REQUEST_RECONF_METRICS, contains all the requested values.
- MSGT_REQUEST_BITSTREAM_SIZES – used by node ROLE_CONTROL to obtain the size of the bitstream for all functions used in the inquired task.
- MSGT_REPLY_BITSTREAM_SIZES – issued as a reply to MSGT_REQUEST_BITSTREAM_SIZES, data field contains pairs: *function_id* and *bitstream_size* in kB implementing *function_id*.
- MSGT_REQUEST_BITSTREAM – sent from node ROLE_PROCESSING to node ROLE_CONTROL to get the bitstream for specified function id.
- MSGT_REPLY_BITSTREAM – reply for the MSGT_REQUEST_BITSTREAM, contains the bitstream for inquired function id.

- MSGT_UNREGISTER_TASK – sent from node ROLE_TASK_OWNER to node ROLE_CONTROL in order to inform the control node that the processing of the task has finished.
- MSGT_DEACTIVATE_BLOCK – sent from node ROLE_TASK_OWNER to all nodes ROLE_PROCESSING that process BLOCK_ONLINE and BLOCK_COOPERATIVE type of blocks. These blocks must be stopped after the task termination is initiated, and all the computations that possibly require data produced by online blocks is no longer required.

- id of a node the object concerns
- number and types of processing units
- ids of functions present on the node
- ids of function implementations present on the node
- ids of datasources, that the node is equipped with.
- *sender addr* – address of the node sending the message
- *recipient addr* – address of the recipient of the message
- *reply to* – contains the address to which the reply should be delivered
- *reply required* – field indicates whether the message requires a reply or not
- *reserved* – field reserved for any use
- *control* – control parameters supported by IIS, e.g. priority
- *msg data* – field containing the data, can be in the form of binary data, text data or marshaled object

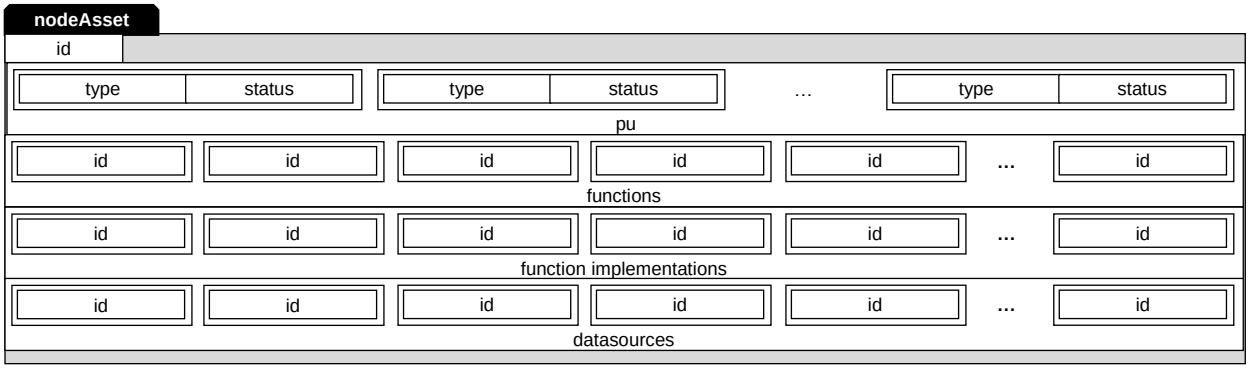


Fig. 27. Architecture of *nodeAssets* object

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeAssets>
        <header>
                <id>#ID</id>
        </header>
        <pu>
                <ts id=1>
                        <type>#TYPE1</type>
                        <status>#STATUS1</status>
                </ts>
                <ts id=2>
                        <type>#TYPE1</type>
                        <status>#STATUS1</status>
                </ts>
                …
                <ts id=tsn>
                        <type>#TYPEn</type>
                        <status>#STATUSn</status>
                </ts>
        </pu>
        <fn>
                <fnid id=1>#FID1</fnid>
                <fnid id=2>#FID2</fnid>
                …
                <fnid id=n>#FIDn</fnid>
        </fn>
        <fi>
                <fiid id=1>#FIID1</fiid>
                <fiid id=2>#FIID2</fiid>
                …
                <fiid id=n>#FIIDn</fiid>
        </fi>
        <ds>
                <dsid id=1>#DSID1</dsid>
                <dsid id=2>#DSID2</dsid>
                …
                <dsid id=n>#DSIDn</dsid>
        </ds>
</nodeAssets>
```

The reliability mechanisms are included too, like the messages timeout, no proper response etc. – to get the reliable experimentation results. These mechanisms provide not only the reliability, but also prevent nodes from flooding with the request messages. Each request is registered at the

originating node and the time is defined that must pass before the next request of the same type can be sent.

Incentives

The incentive mechanisms should be always considered in the distributed processing system – to control the willingness of nodes to contribute and perform the processing of blocks. The incentive mechanisms are not implemented as this is beyond the scope of this work, however operational space is reserved in the node. This way, it is easy to add the appropriate incentive mechanism that would focus on the goals specified.

Fully decentralized systems (DHT)

The communication is based on control nodes as the primary approach. The other approach is the fully decentralized approach based on DHT (Distributed Hash Table). Such a mechanism allows to ask the system for asset location, without the need to ask for any specific node (like the control node in the primary approach). DHT approach has been widely used in many applications. It is a promising approach for decentralized systems [PZ07], [Zha04], and many authors have implemented this approach to achieve system decentralization [CL13]. Authors of [UQ05] implemented DHT, together with a Common Object Request Broker Architecture (CORBA) standard, in order to allow the overlay-network-based communication system to conduct event management. Attempts have been made to provide a DHT-based approach to wireless sensor networks; for example, the Pastry algorithm was used to design a data sharing system for the P2P sensor structure by the authors of [SKS09]. The researchers used DHT Pastry to provide the current

sensor data; however, PAST algorithm they proposed also can access historical sensor values. Applications of distributed systems using DHT include vehicular ad hoc network [DMI11], peer-to-peer data sharing networks [TCC11], web caching [RHM12], instant messaging, and many others. Despite significant advantages in a fully decentralized system, the architecture presented in this work does not implement full decentralization. The behavior of DHT used for distributed computation is shown in [CL13]. However, the proposed architecture could be easily adapted for DHT: only the locations discovery system would have to be replaced in such case.

4.4. Operation of DPRS

The operation of DPRS has an object-based nature. This way each system component is considered as an object with properties, internal architecture and associated algorithms. Therefore, the system operation might be described object-by-object, along with the interconnection capabilities among them.

4.3.1. General operation of the DPRS

Logically, the whole DPRS system is considered as one structure (Fig. 1) and can be described using a workflow even though its components are operating autonomously (Fig. 28). System begins its life in the DP_PHASE_INIT stage, with at least one node. The nodes in the initial group, first create an entry point that will be accessible for further nodes joining the system. In the next step, new nodes access the entry point and proceed to join the system. Nodes can freely join the system at this step, but also anytime later, during regular (non-initial) system operation. Next, the control node(s) are designated using the algorithm AL_DESIGNATE_CONTROL_NODES. After this point, the DPRS starts its regular processing part. However, nodes can join and exit the system, and AL_DESIGNATE_CONTROL_NODES is used for the system reconfiguration during the phase of roles reassignment (if such phase is included, as it is not mandatory).

During the processing stage – DP_PHASE_OPERATION, nodes that bear the role ROLE_TASK_OWNER register the tasks to the node with ROLE_CONTROL assigned. Multiple tasks can operate at the same time. When the task is registered, it becomes available for other nodes for processing. The processing is done in a way, that nodes with ROLE_PROCESSING role request block for the processing and perform the processing in a way depending on the type of the block. If the result of block processing produces the result right after processing is finished, then the result must be disseminated to the designated receivers. Such dissemination occurs for

BLOCK_OFFLINE and BLOCK_DEFINED. Periodically, the DPRS performs the reconfiguration phase. During this part, the roles might be reassigned using AL_REASSIGN_ROLES (including the control ROLE_CONTROL), also entry point might be reassigned. Task processing (processing blocks) lasts until the task's end condition is satisfied. In such case, the task is deregistered at the node ROLE_CONTROL, becoming inactive for processing nodes. Also, currently running blocks of type BLOCK_ONLINE and BLOCK_COOPERATIVE are terminated – by the termination message MSGT_UNREGISTER_TASK issued by node ROLE_TASK_OWNER, sent along with MSGT_DEACTIVATE_BLOCK used to terminate online and cooperative blocks. DPRS system continues the operation with the other tasks(s) until the DPRS end condition is satisfied.

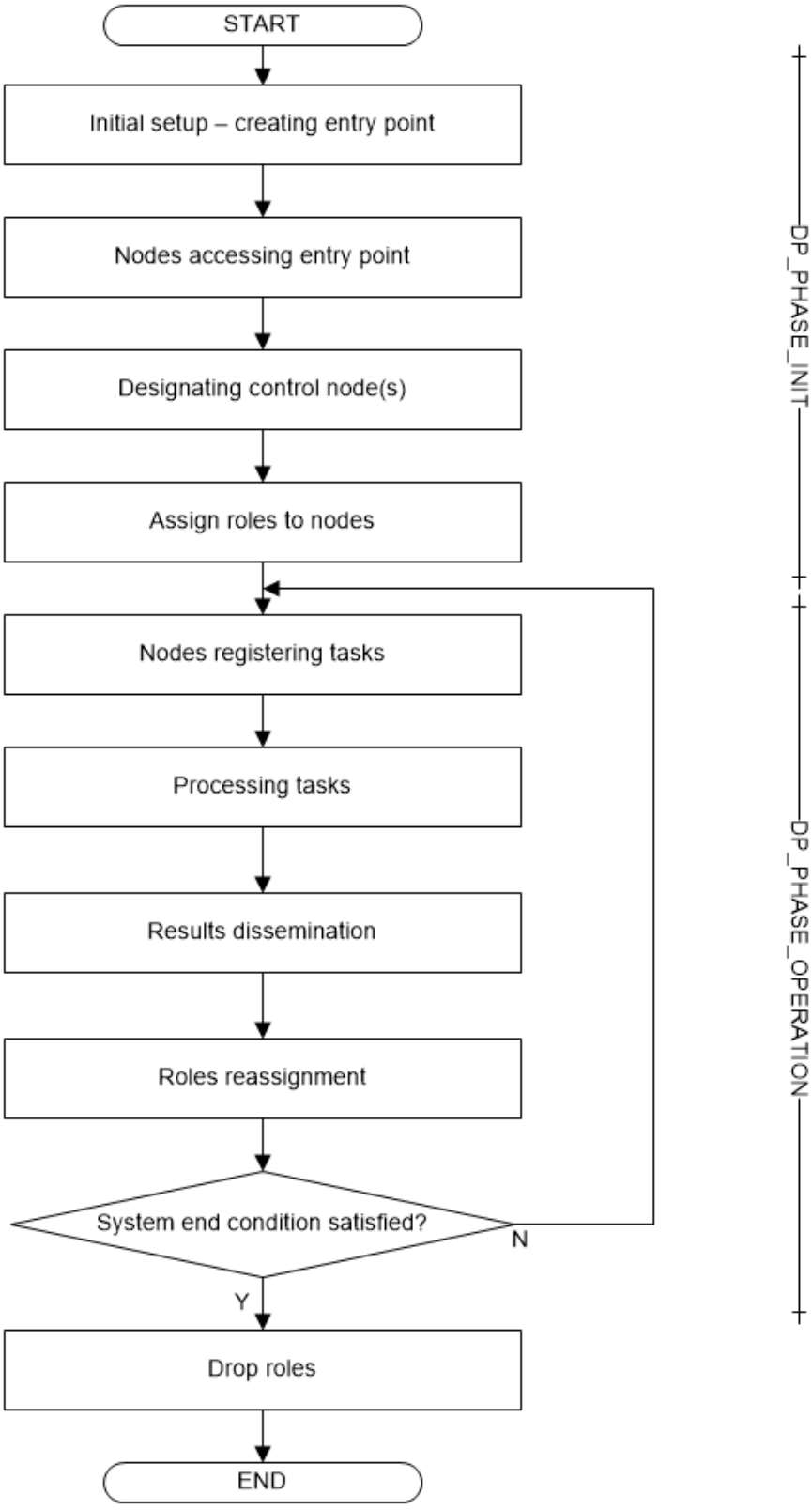


Fig. 28. General workflow of DPRS system for a single task

4.3.2. Operation of the node

Each node runs the algorithm AL_NODE that contains multiple reconfigurable elements that might be replaced even during runtime. Node is supplied with multiple configuration entries (the *header* section of the XML structure). The only required entry is the entry node *enode*. During the NODE_PHASE_INIT, the value(s) of *enode* are used to locate the node that is a ROLE_CONTROL and is able to provide suitable information for the node to complete the NODE_PHASE_INIT. If the system is in DP_PHASE_INIT (Fig. 28), then the newly joining node will be involved in roles assignment (AL_DESIGNATE_CONTROL_NODES). After the NODE_PHASE_INIT is completed, node enters the NODE_PHASE_SCHEDULER during which the events scheduled prior to executing tasks are processed. These events may include various actions, such as registering task, unregistering task, leaving the system, etc. Next is the NODE_PHASE_ROLES phase. During this phase, the algorithms associated with each role are executed. Then the control can loop again to the NODE_PHASE_SCHEDULER, or finish the processing. The general diagram of the *AL_NODE* is depicted in Fig. 29. The *execute roles* box in the NODE_PHASE_ROLES contains one or more role workflows, associated with roles – depending how many roles a node bears. The diagram in Fig. 21 does not include the reconfiguration phase NODE_PHASE_RECONF, as it is included in the ROLE_RECONFIGURABLE role.

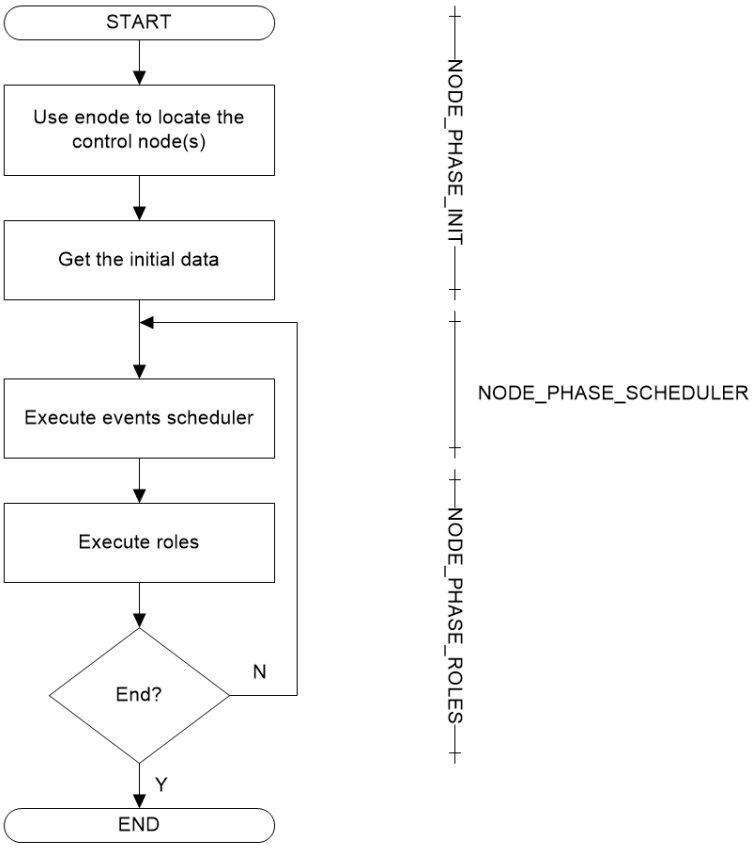


Fig. 29. Top-level scheme for node phases (AL_NODE)

Roles of the nodes

DPRS system allows multiple roles in the system with algorithm defined for each of it. However, the minimal set of roles must be implemented in the system: ROLE_BASIC, ROLE_CONTROL, ROLE_TASK_OWNER and ROLE_PROCESSING. These roles provide the basic operation of the system: distributed processing with autonomous nodes. The additional ROLE_RECONFIGURABLE is assigned to all the processing units that hold ROLE_PROCESSING role and have the reconfigurable FPGA chip installed. The workflow for these roles is as follows:

ROLE_BASIC

Operation of the ROLE_BASIC is shown in Fig. 30. This operation is not looped – it is performed at the node initialization – during the NODE_PHASE_INIT. Messages involved:

- MSGT_REQUEST_CNTNODES – each node might be asked for its knowledge about nodes ROLE_CONTROL. Which node is asked is determined by enode. Node *v* replies with the MSGT_REPLY_CNTNODES.
- MSGT_REPLY_CNTNODES – node *v* receives the list of node identifiers that are known to the issuer of MSGT_REPLY_CNTNODES as the ones assigned ROLE_CONTROL. All of the identifiers are added to node's *control nodes* field.

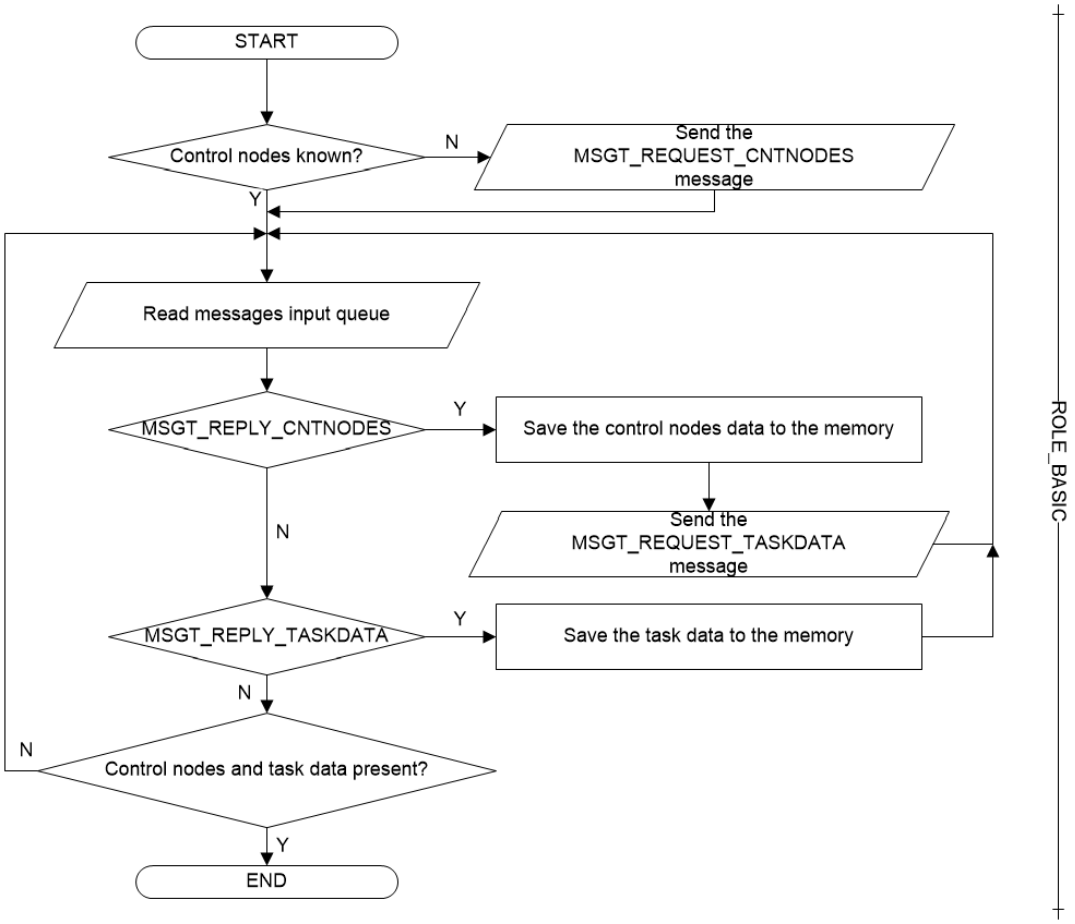


Fig. 30. Operation of ROLE_BASIC

ROLE_CONTROL

The operation required of this role includes responding to messages received from other nodes. Processing of a message depends on the message's type. Fig. 31 presents the workflow for the ROLE_CONTROL. It also shows the minimal set of messages that must be accomplished by ROLE_CONTROL of node *v* implementing this role:

- MSGT_REGISTER_TASK – node *v* adds the task information to its register, along with the sender node *w* of the message that is assigned the role of ROLE_TASK_OWNER. The status of the task is now set as STATUS_TASK_REGISTERED at node *v* and at the node with the reception of MSGT_ACK sent by node *v*.
- MSGT_REQUEST_TASK_DATA – node *v* sends the digest information MSGT_REPLY_TASK_DATA about tasks that are currently registered, to the requesting node *w*. The minimal dataset includes: task id, task owner and block counts, except BLOCK_OFFLINE. After issuing the message, node *v* reads the NODE_ASSET structures embedded in the MSGT_REQUEST_TASK_DATA and stores in the local memory.
- MSGT_REQUEST_DS_LOCATIONS – node *v* sends the MSGT_REPLY_DS_LOCATIONS message containing all the known locations for each data source. It might include the information that the provided list is complete.
- MSGT_REQUEST_BITSTREAM_SIZES – node *v* replies with the MSGT_REPLY_BITSTREAM_SIZES containing the bitstream sizes for the FPGA with their corresponding id:s.
- MSGT_REQUEST_BITSTREAM – node *v* replies with the MSGT_REPLY_BITSTREAM with the implemented function, which is the FPGA bitstream.

- MSGT_UNREGISTER_TASK – node *v* removes the task of its id included in the message and removes the data related to this task.

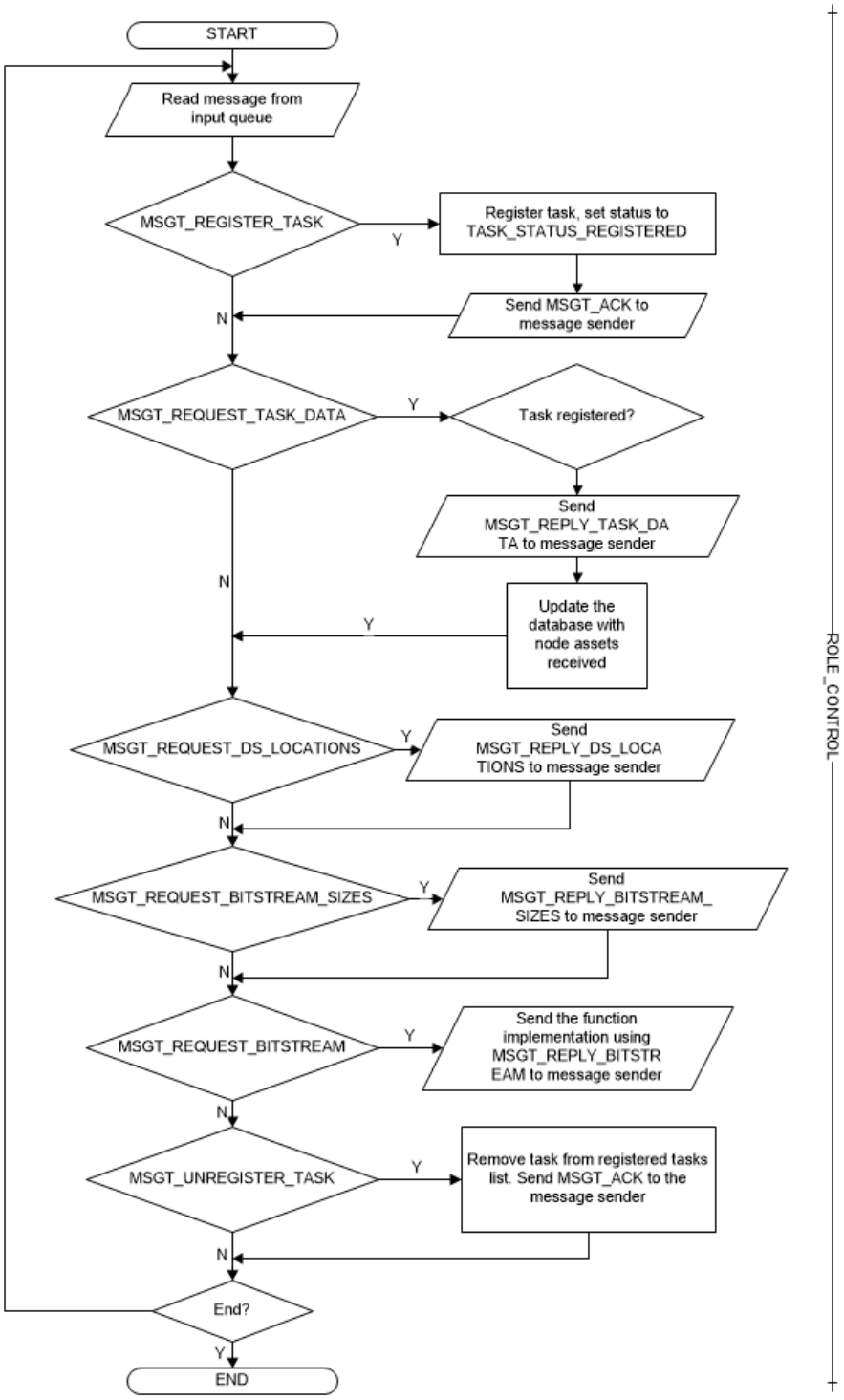


Fig. 31. Node workflow for ROLE_CONTROL

Nodes ROLE_CONTROL synchronize their knowledge bases between each other every defined period of time.

ROLE_PROCESSING

The functions assigned to this role are present in multiple stages, presented in Fig. 32.

First stage is a NODE_PHASE_PROCESSING_CONFIGURATION. This stage includes the initial configuration of the node, and possible configuration adjustments in further executions of the ROLE_PROCESSING algorithm. Among many aspects, a node configures its knowledge regarding the control nodes ROLE_CONTROL, adjusts the information regarding the computing tasks available, and configures itself for the computation process.

The second stage is NODE_PHASE_PROCESSING_REQUESTING during which a node decides about getting further blocks for processing, depending on the resources available and node predicting algorithms. The third stage is a NODE_PHASE_PROCESSING_BLOCKS and includes all the actions within the block processing. The fourth stage is NODE_PHASE_PROCESSING_MSGS during which the node replies to the messages it has in its processing queue. The following messages received by node *v* are involved in this phase and role:

- MSGT_REPLY_TASK_DATA – the ROLE_PROCESSING node receives the details about task(s) from the ROLE_CONTROL node. The essential data is a pair (*z*, *w*), where *w* is the identifier (translatable to a network address) of the node that is issuer (ROLE_TASK_OWNER) of the task *z*.

- MSGT_FUNCTIONS_ATT – message contains function implementation that is saved to node's *v* local memory.
- MSGT_REPLY_BLOCK – message contains the marshaled block *b* object. After receiving this message, the data field is unmarshaled and the object is created. The block's status is set to STATUS_BLOCK_IDLE and is added to the blocks section of the node structure. The object data saved locally on the requesting node is read at this point and used to match the request with the received block.
- MSGT_DEACTIVATE_TASK – after receiving this message, the node *v* removes the task id *z* referred in the message's body from its list of registered tasks, by scheduling the appropriate event (EVT_DEACTIVATE_TASK).

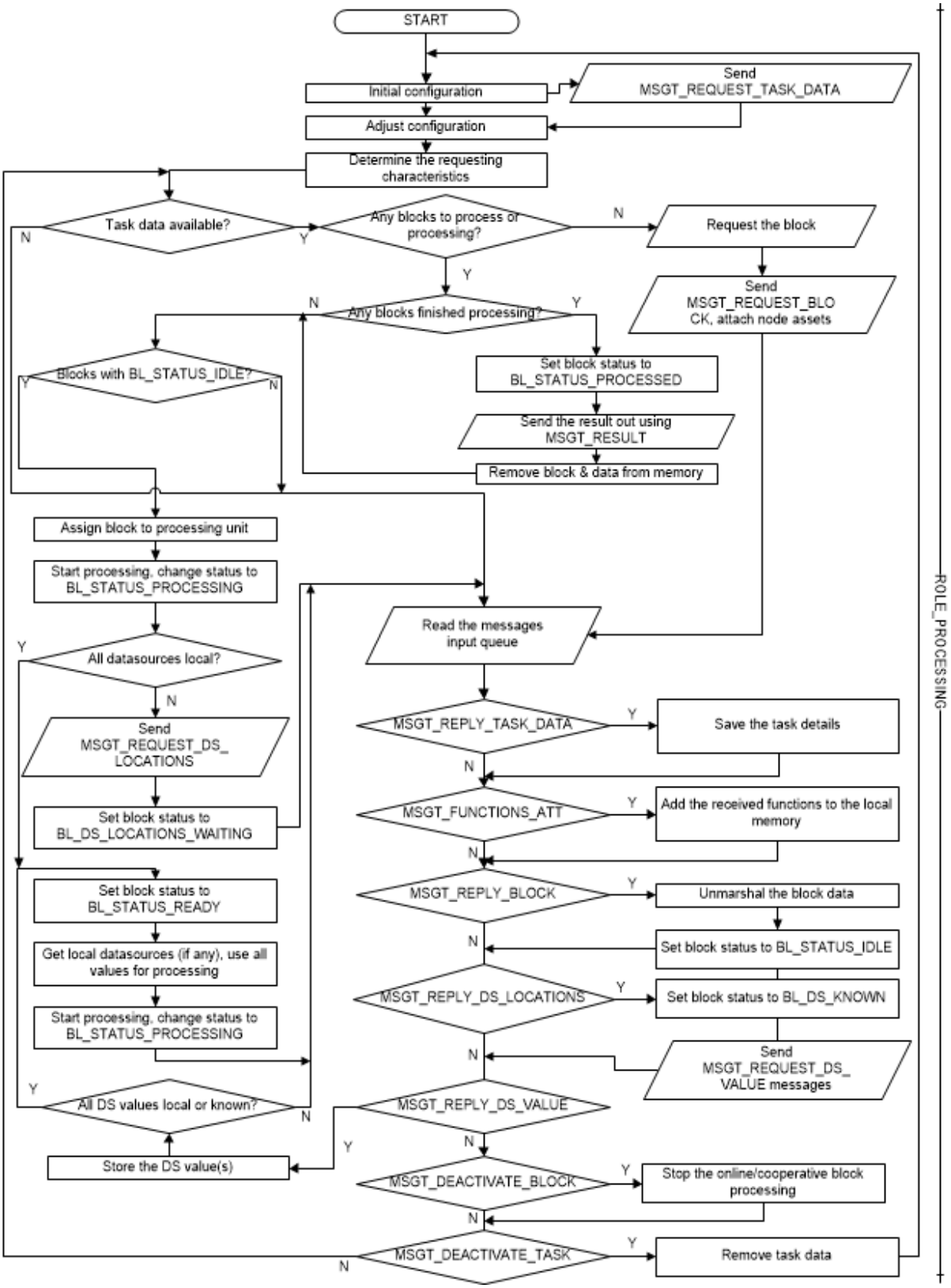


Fig. 32. ROLE_PROCESSING processing stages

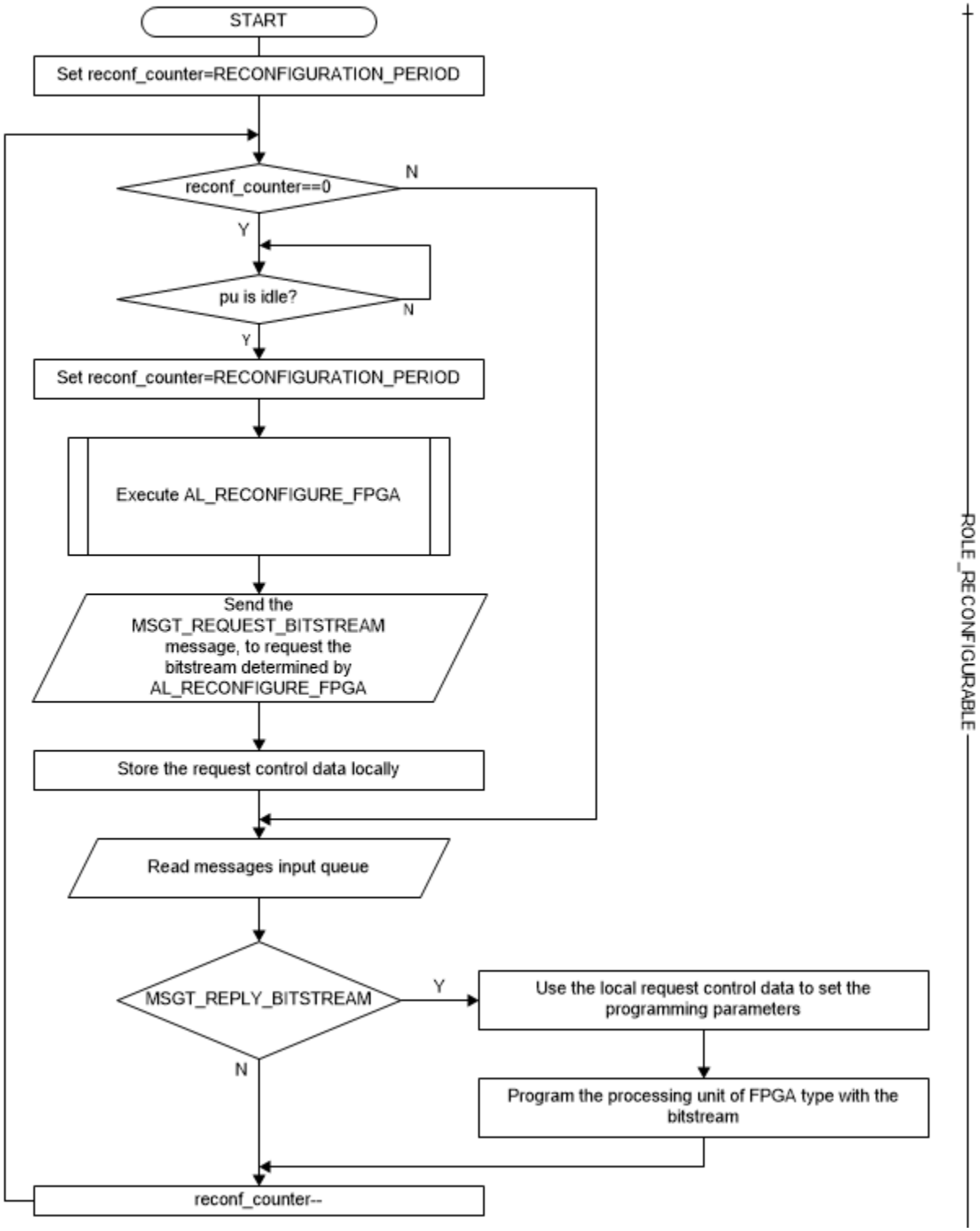


Fig. 33. Reconfiguration scheme (ROLE_RECONFIGURABLE)

The reconfiguration scheme presented in Fig. 33 is executed on every processing unit that has the reconfigurability capabilities. It runs concurrently with the operation of ROLE_PROCESSING, however these two operations rely on each other – the ROLE_RECONFIGURABLE can only perform the reconfiguration once the processing unit is not performing any computation (i.e. is not occupied with any block). Therefore, as processing units process blocks of various sizes and it takes them different time to process, the processing units are reprogrammed at various moments of time. Separate ROLE_RECONFIGURABLE operation is run for each processing unit.

ROLE_RECONFIGURABLE

This role is assigned to each node/processing unit that has a reconfigurable FPGA element and has ROLE_PROCESSING. The operation of this role is presented in Fig. 33. The reconfiguration mechanism uses the *reconf_counter* that counts in the cycles set by RECONFIGURATION_PERIOD, and is decremented every time slot. Once the *reconf_counter* reaches zero (that is, each RECONFIGURATION_PERIOD slots), the algorithm checks if the reconfiguration is desired by executing AL_RECONFIGURE_FPGA. If the processing unit is busy with processing while *reconf_counter* reaches zero, then the algorithm waits until the processing is finished – *reconf_counter* does not go below zero. Once the AL_RECONFIGURE_FPGA is executed, it determines if reconfiguration should be done at that time, and also which function should be programmed into the processing unit executing the AL_RECONFIGURE_FPGA. If reconfiguration is decided, then the MSGT_REQUEST_BISTREAM message is sent to the ROLE_CONTROL and the related data object is stored locally, so the incoming bitstream will be properly matched with the requesting

processing unit and remaining parameters can be set. Just before executing AL_RECONFIGURE_FPGA the *reconf_counter* is set to RECONFIGURATION_PERIOD.

ROLE_TASK_OWNER

This role is assigned to the node that inputs the task into the system, for further processing. The operation is described in Fig. 34. The following messages are involved:

- MSGT_REQUEST_BLOCK – node *v* receives this message as a block request with or without the supplementary information about the node's current state. Based on the supplied information, the block matching algorithm AL_MATCH_BLOCK_TO_PU selects the most appropriate block for the node *w* that issues the MSGT_REQUEST_BLOCK message. Also, the additional matching parameters might be provided by node *w*, such as strict matching, request for a specific block type, etc. Node *v* responds to node *w* with the MSGT_REPLY_BLOCK message enclosing the marshaled block *b* object. The system configuration might also run the automatic functions sendout – in such case, the node *v* will issue MSGT_FUNCTIONS_ATT, that include the function(s) implementations required to process the block enclosed in the MSGT_REPLY_BLOCK sent along. Only function implementation(s) that are missing on the node *w* are sent by MSGT_FUNCTIONS_ATT message(s). In case no blocks are available to send, node *v* sends MSGT_INFO_NOMOREBLOCKS to requesting node *w*.

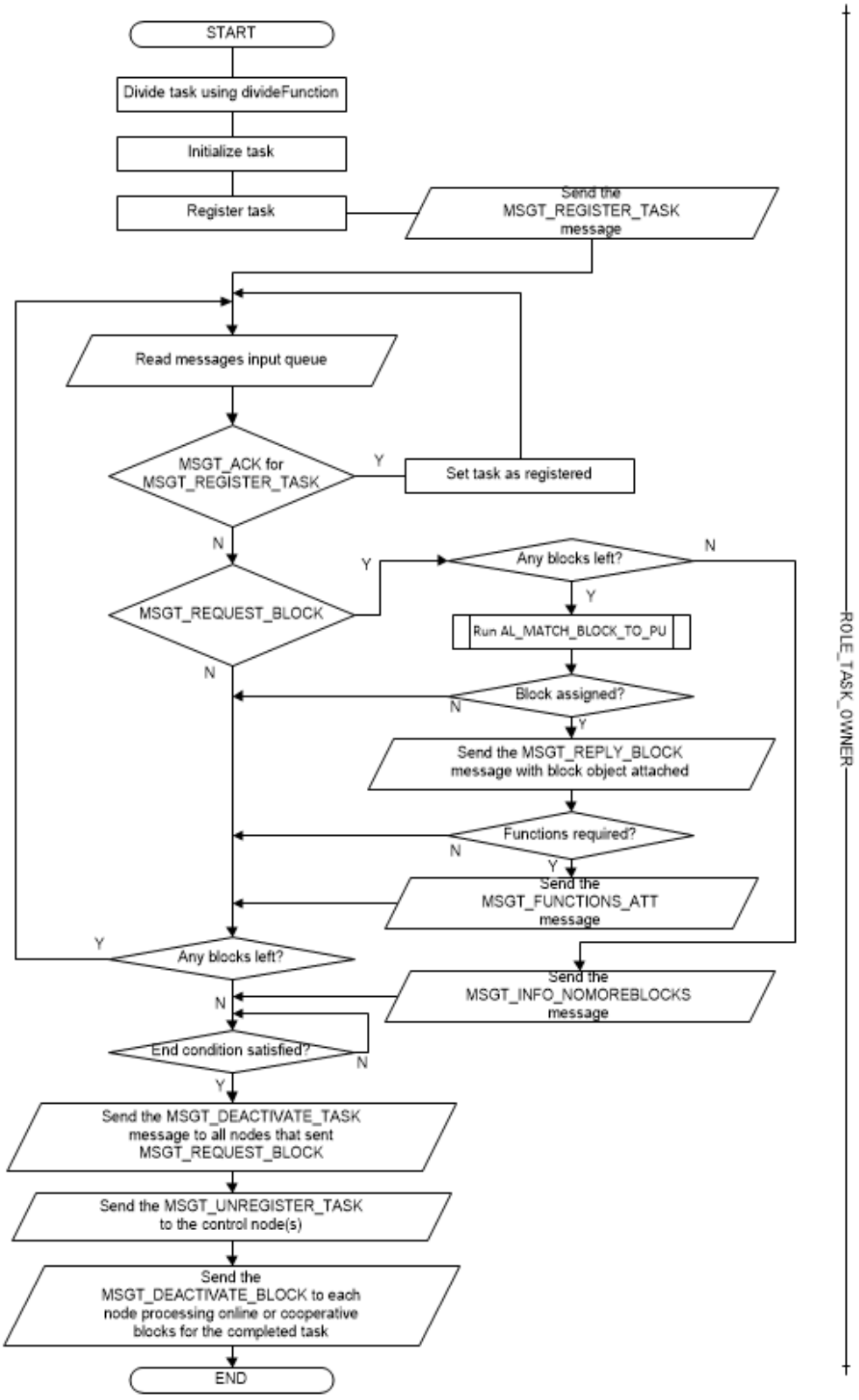


Fig. 34. Operation of ROLE_TASK_OWNER

4.5. Operational algorithms

This section describes the algorithms that are designed for the DPRS and are used to run the system. Thanks to the modular architecture of the system, each of these algorithms can be replaced with another one.

4.5.1. AL_RECONFIGURE_FPGA

AL_RECONFIGURE_FPGA algorithms are performed at the node with ROLE_PROCESSING and ROLE_RECONFIGURABLE roles. In this process, such node requests specific data from ROLE_CONTROL node that collects the data regarding the nodes. Data is collected by ROLE_PROCESSING nodes using reconfiguration information sent by nodes along with the requests (including their local resources, state, etc.). They also communicate with nodes ROLE_TASK_OWNER to maintain current blocks' statistics. AL_RECONFIGURE_FPGA yields the id of the function (f_{cand}) that is decided to be programmed on to the FPGA.

AL_RECONFIGURE_FPGA_1

The idea of AL_RECONFIGURE_FPGA_1 is an even distribution of functions over the system nodes. The experiments are assumed to show how the even availability of functions impact the overall efficiency of the system. The steps of the algorithm are as follows, also depicted in Fig. 35.

Algorithm: AL_RECONFIGURE_FPGA_1

1. Request the reconfiguration metrics from ROLE_CONTROL
   1.1. Attach the current information about the node and pu:s
2. The node v requesting reconfiguration metrics for task z from ROLE_CONTROL node sets the f_{cand} as:

$$f_{cand} \rightarrow f_{next,z}$$

where $f_{next,z}$ determines the next function compared to previously assigned f_{cand}, that is

$$f_{next,z}: \sum_b^B x_{b,z} n_{b,f_{next}} > 0$$

3. Send the information MSGT_RECONF_STATUS to the ROLE_CONTROL about programmed function.

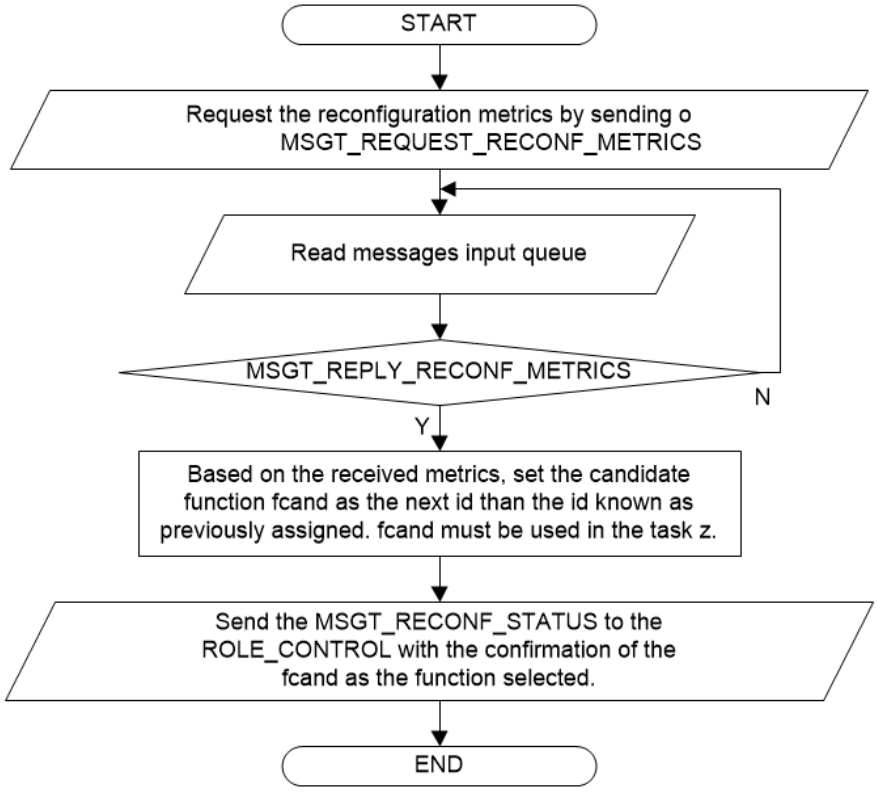


Fig. 35. The operation of AL_RECONFIGURE_FPGA_1

AL_RECONFIGURE_FPGA_2

The idea of AL_RECONFIGURABLE_FPGA_2 is to program the function that is the sole requirement for most of the unprocessed blocks. The operation is shown in Fig. 36.

Algorithm: AL_RECONFIGURE_FPGA_2

1. Request the reconfiguration metrics from ROLE_CONTROL
   1.1. Attach the current information about the node and pu:s

2. The node v requesting reconfiguration metrics for task z (active and considered task) from ROLE_CONTROL node, calculate the Φ_f value for each function f:

$$\Phi_f = \sum_b^B x_{b,z} n_{b,f}$$

where b status is *STATUS_IDLE*

where $\sum_f^F n_{b,f} = 1$

3. Select $f_{cand} \rightarrow \forall \Phi_f : \Phi_{f_{cand}} \geq \Phi_f$
4. Send the information MSGT_RECONF_STATUS to ROLE_CONTROL for function to be programmed.

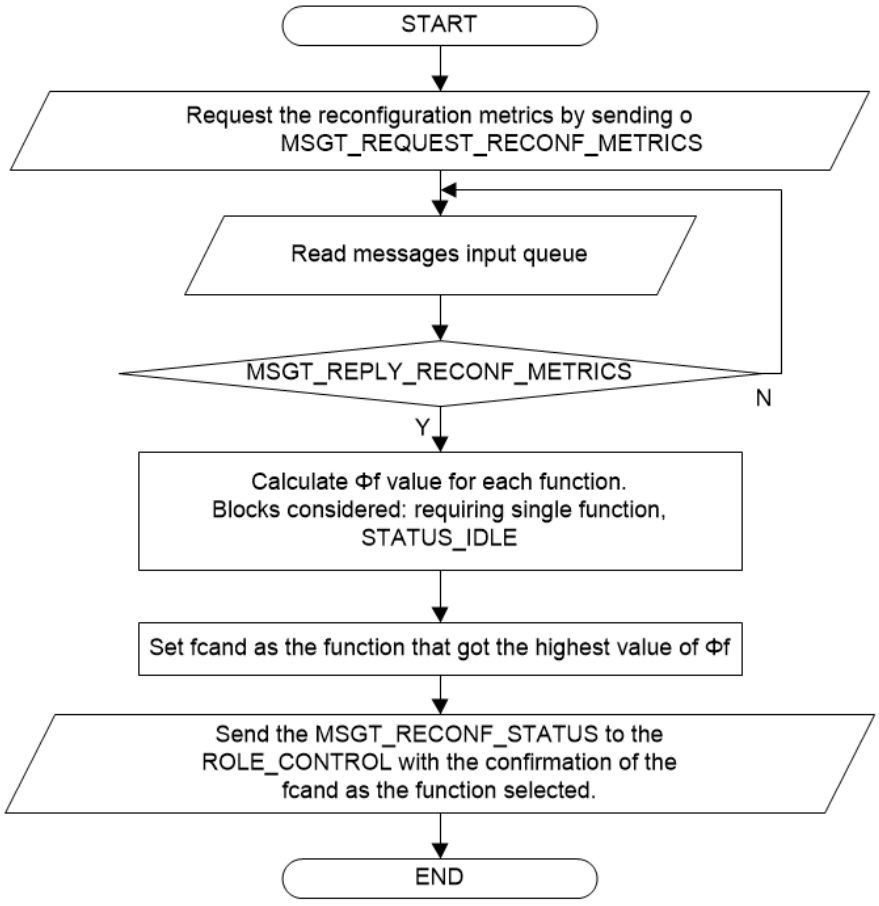


Fig. 36. The operation of AL_RECONFIGURE_FPGA_2

AL_RECONFIGURE_FPGA_3

Algorithm AL_RECONFIGURE_FPGA_3 extends the AL_RECONFIGURE_FPGA_2, and also leads to programing the function that is the sole requirement for most of the unprocessed blocks, but also matches the datasources available on node *v* with the datasources that are available on the nodes that satisfy the function requirement (Fig. 37).

Algorithm: AL_RECONFIGURE_FPGA_3

1. Request the reconfiguration metrics from ROLE_CONTROL
   1.1. Attach the current info about the node and pu:s
2. The node *v* requesting reconfiguration metrics for task *z* (active and considered task) from ROLE_CONTROL node, calculate the Φ_f value for each function *f*:

$$\Phi_f = \sum_b^B \sum_d^D x_{b,v} n_{b,f} k_{v,d} l_{b,d}$$

where b status is *STATUS_IDLE*

where $\sum_f^F n_{b,f} = 1$

3. Select $f_{cand} \rightarrow \forall \Phi_f : \Phi_{f_{cand}} \geq \Phi_f$
4. Send the information MSGT_RECONF_STATUS to the ROLE_CONTROL about programmed function.

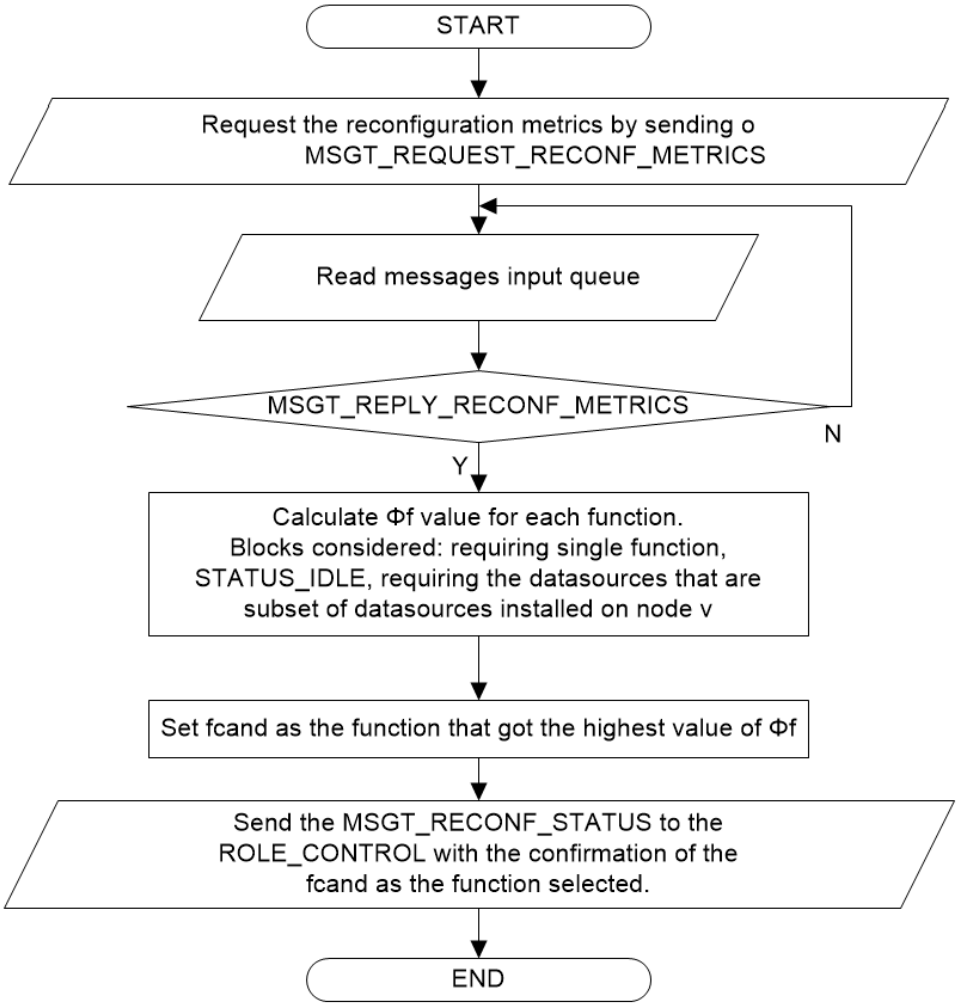


Fig. 37. Operation of AL_RECONFIGURE_FPGA_3

4.5.2. AL_MATCH_BLOCK_TO_PU

The AL_MATCH_BLOCK_TO_PU algorithm is executed on the ROLE_TASK_OWNER in response to the MSGT_REQUEST_BLOCK along with the *nodeAssets* object attached. Based on the local knowledge of the ROLE_TASK_OWNER and the *nodeAssets*, the ROLE_TASK_OWNER matches the block to the requesting node, specifically to one of its processing units – as the MSGT_REQUEST_BLOCK is related to the processing unit. Algorithms have to consider FPGA processing units separately, as the number *FP* of programmed functions *{FS}* must be included.

AL_MATCH_BLOCK_TO_PU_1

This algorithm is allocating blocks online first, then cooperative blocks, and then defined and offline blocks subsequently, in order of their appearance in the ROLE_TASK_OWNER structures (Fig. 38).

Algorithm: AL_MATCH_BLOCK_TO_PU_1

```
1. if online blocks available
     Analyze nodeAssets
     Set {BU} -> unallocated blocks online, that:
           Datasources required by block = datasources present on
           node
2. else if cooperative blocks available
     Set {BU} -> unallocated blocks cooperative
3. else if defined blocks available
      Set {BU} -> unallocated blocks defined
4. else if offline blocks available
      Set {BU} -> unallocated blocks offline
5. if processing unit type = FPGA, then remove from {BU} all
   blocks, that:
     number of functions required by block > FP
     {FS} is not a (sub)set of functions required by block
5. matched_block <- BU.first_element
6. if matched_block assigned
     return matched_block
   else return null
```

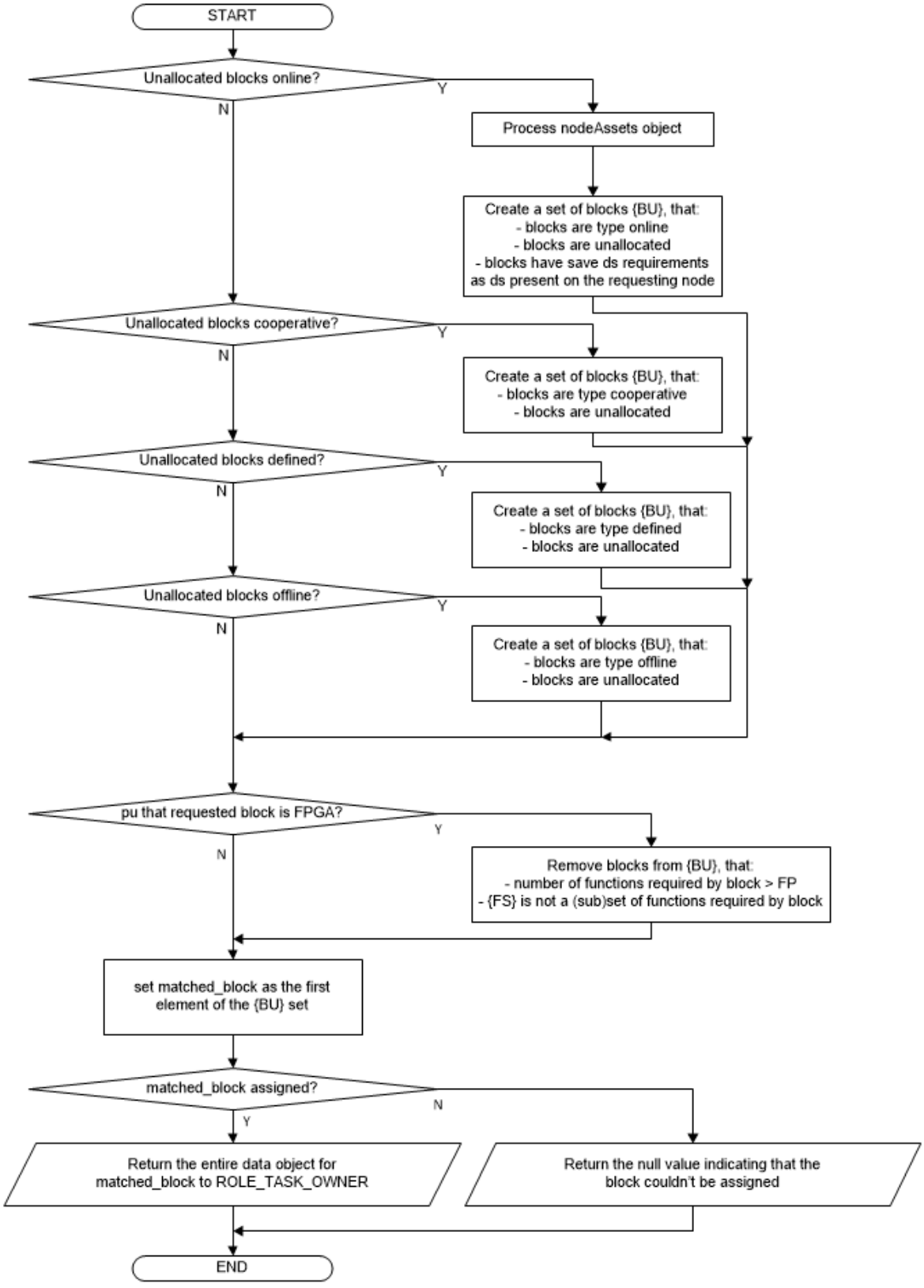


Fig. 38. Operation of AL_MATCH_BLOCK_TO_PU_1 algorithm

AL_MATCH_BLOCK_TO_PU_2

This algorithm considers datasources required by blocks and matches them to nodes' local datasources. The operation is explained below in steps and as diagram (Fig. 39).

Algorithm: AL_MATCH_BLOCK_TO_PU_2

```
1. if online blocks available
     Analyze nodeAssets
     Set {BU} -> unallocated blocks online, that:
           Datasources required by block = datasources present on
           node
2. else if cooperative blocks available
     Set {BU} -> unallocated blocks cooperative, that:
           Datasources required by block = datasources present on
           node
3. else if defined blocks available
     Set {BU} -> unallocated blocks defined, that:
           Datasources required by block = datasources present on
           node
4. else if offline blocks available
     Set {BU} -> unallocated blocks offline, that:
           Datasources required by block = datasources present on
           node
5. if processing unit type = FPGA, then remove from {BU} all
   blocks, that:
     number of functions required by block > FP
     {FS} is not a (sub)set of functions required by block
5. matched_block <- BU.first_element
6. if matched_block assigned
     return matched_block
   else return null
```

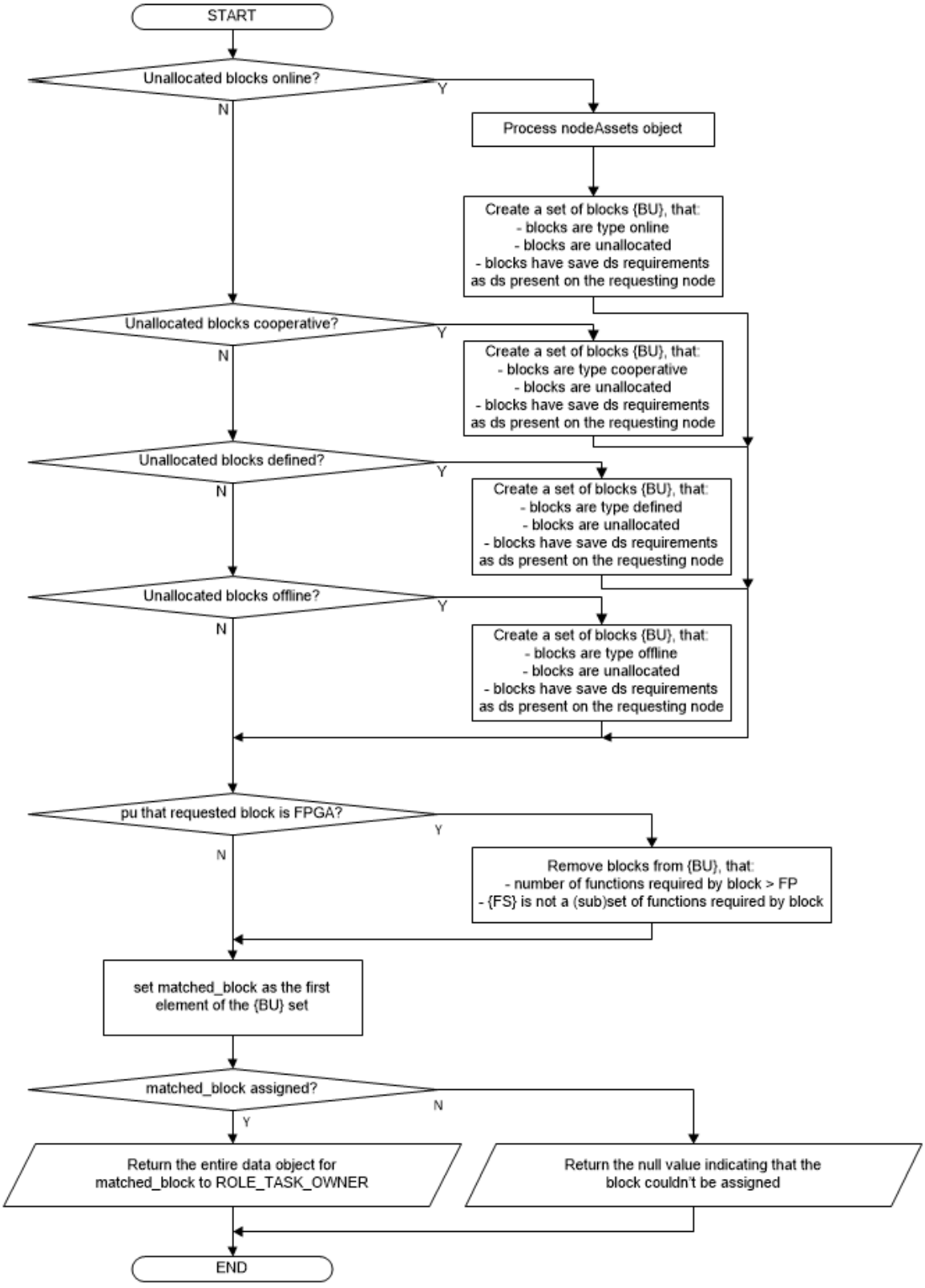


Fig. 39. Operation of AL_MATCH_BLOCK_TO_PU_2 algorithm

AL_MATCH_BLOCK_TO_PU_3

This algorithm operates on the virtual ranges that are defined for both nodes and blocks. Blocks are assigned from the same range as the node. The criterion ND_CRITERION can be defined so the efficiency of the algorithm can be adjusted for specific systems. The below steps describe the operation of the AL_MATCH_BLOCK_TO_PU_3 algorithm, and the flow is shown in Fig. 40.

Algorithm: AL_MATCH_BLOCK_TO_PU_3

```
1. if online blocks available
          Analyze nodeAssets
          Set {BU} -> unallocated blocks online, that:
               Datasources required by block = datasources
               present on node
          go to step 15
   else if cooperative blocks available
          Set {BU} -> unallocated blocks cooperative
          go to step 15
2. Set CR (current_range) to 0
3. Create array BU containing all unallocated blocks defined
4. Create array ND containing all the nodes (known to
   ROLE_TASK_OWNER node) that bear role ROLE_PROCESSING
5. Calculate number of ranges: rng = BU.size / ND.size
6. rng = rng - CR
7. if rng < 0 then rng = 1
8. Sort BU by block size
9. Sort ND by ND_CRITERION
10. Calculate the range size for BU: BU_RS = max{1, ⌊BU.size / rng⌋}
11. Calculate the range size for ND: ND_RS = max{1, ⌊ND.size / rng⌋}
12. For requesting node v, determine to which ND range v_range
```

```
    it belongs and assign: matched_block <- BU.v_range.first
13. If matched_block = none, then:
          if rng > 1
               CR = CR + 1
               go to step 6
    else
          go to step 17
14. if matched_block = none && if offline blocks available
          Set {BU} -> unallocated blocks offline
15. if processing unit type = FPGA, then remove from {BU} all
    blocks, that:
          number of functions required by block > FP
          {FS} is not a (sub)set of functions required by block
16. matched_block <- {BU}.first_element
17. if matched_block assigned
          return matched_block
    else return null
```

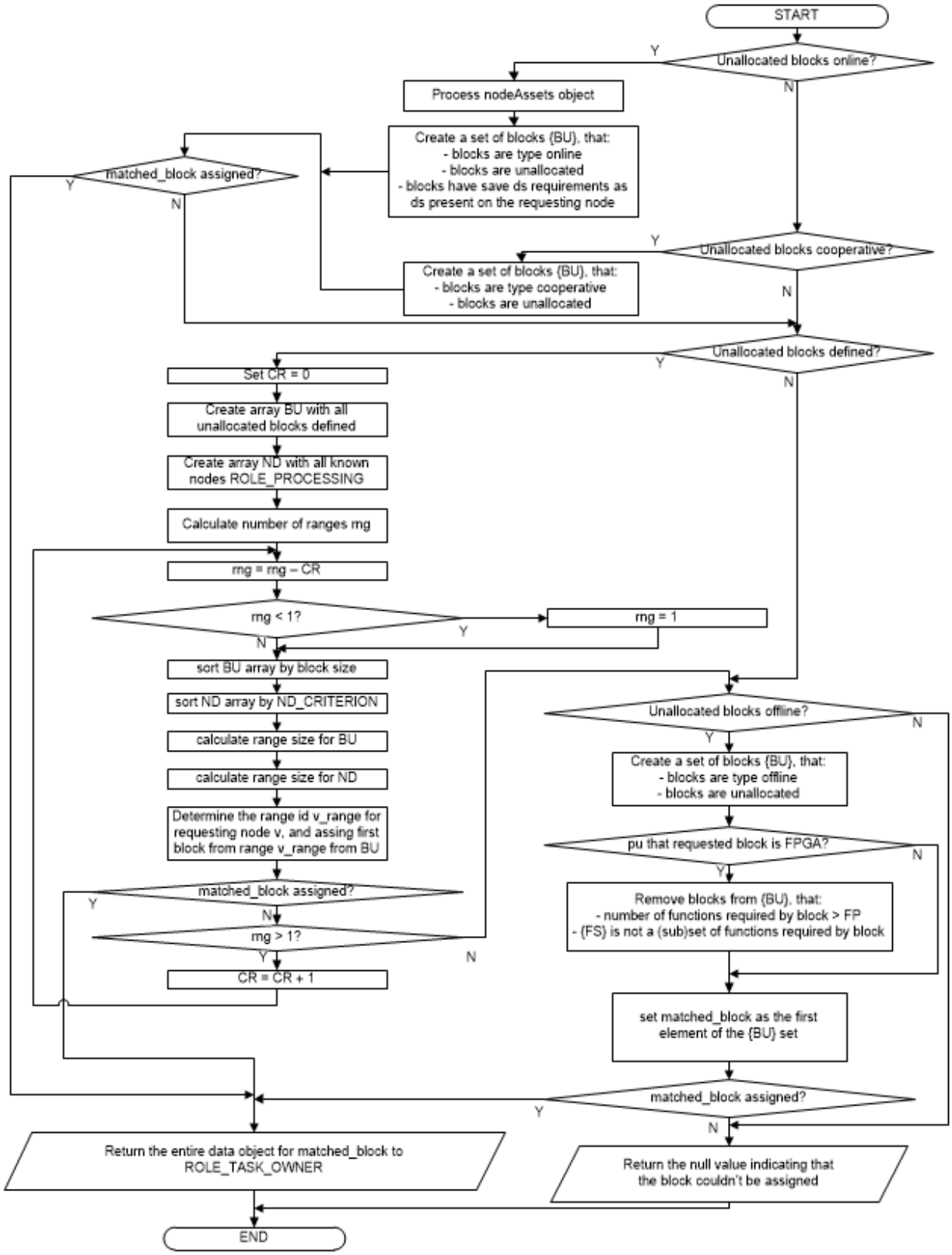


Fig. 40. Operation of AL_MATCH_BLOCK_TO_PU_3 algorithm

AL_MATCH_BLOCK_TO_PU_4

This algorithm operates on virtual ranges that are defined for both nodes and blocks – similar to the AL_MATCH_BLOCK_TO_PU_3. Here, only blocks defined that have matching datasources with the requesting node are considered in the BU array. Also, when the processing unit is a type of reprogrammable FPGA, then in BU array only the blocks that require function(s) that already are programmed in the requesting pu are considered (Fig. 41).

Algorithm: AL_MATCH_BLOCK_TO_PU_4

1. if online blocks available
   Analyze *nodeAssets*
   Set {BU} -> unallocated blocks online, that:
   Datasources required by block = datasources present on node
   go to step 15
   else if cooperative blocks available
   Set {BU} -> unallocated blocks cooperative
   go to step 15
2. Set CR (current_range) to 0
3. Create array BU containing all unallocated blocks defined, that:
   Datasources required by block = datasources present on node
   if requesting pu is FPGA, then include only blocks that require function(s) already programmed in pu
4. Create array ND containing all the nodes (known to ROLE_TASK_OWNER node) that bear role ROLE_PROCESSING
5. Calculate number of ranges: $rng = \frac{BU.size}{ND.size}$
6. rng = rng – CR
7. if rng < 0 then rng = 1

8. Sort BU by block size
9. Sort ND by ND_CRITERION
10. Calculate the range size for BU: $BU\_RS = \max\left\{1, \left\lfloor \frac{BU.size}{rng} \right\rfloor\right\}$
11. Calculate the range size for BU: $ND\_RS = \max\left\{1, \left\lfloor \frac{ND.size}{rng} \right\rfloor\right\}$
12. For requesting node v, determine to which ND range v_range it belongs and assign: matched_block <- {BU}.v_range.first
13. If matched_block = none, then:

```
        if rng > 1
            CR = CR + 1
            go to step 6
    else
        go to step 17
```

14. if matched_block = none && if offline blocks available

```
        Set {BU} -> unallocated blocks offline
```

15. if processing unit type = FPGA, then remove from {BU} all blocks, that:

```
        number of functions required by block > FP
        {FS} is not a (sub)set of functions required by block
```

16. matched_block <- {BU}.first_element
17. if matched_block assigned

```
        return matched_block
    else return null
```

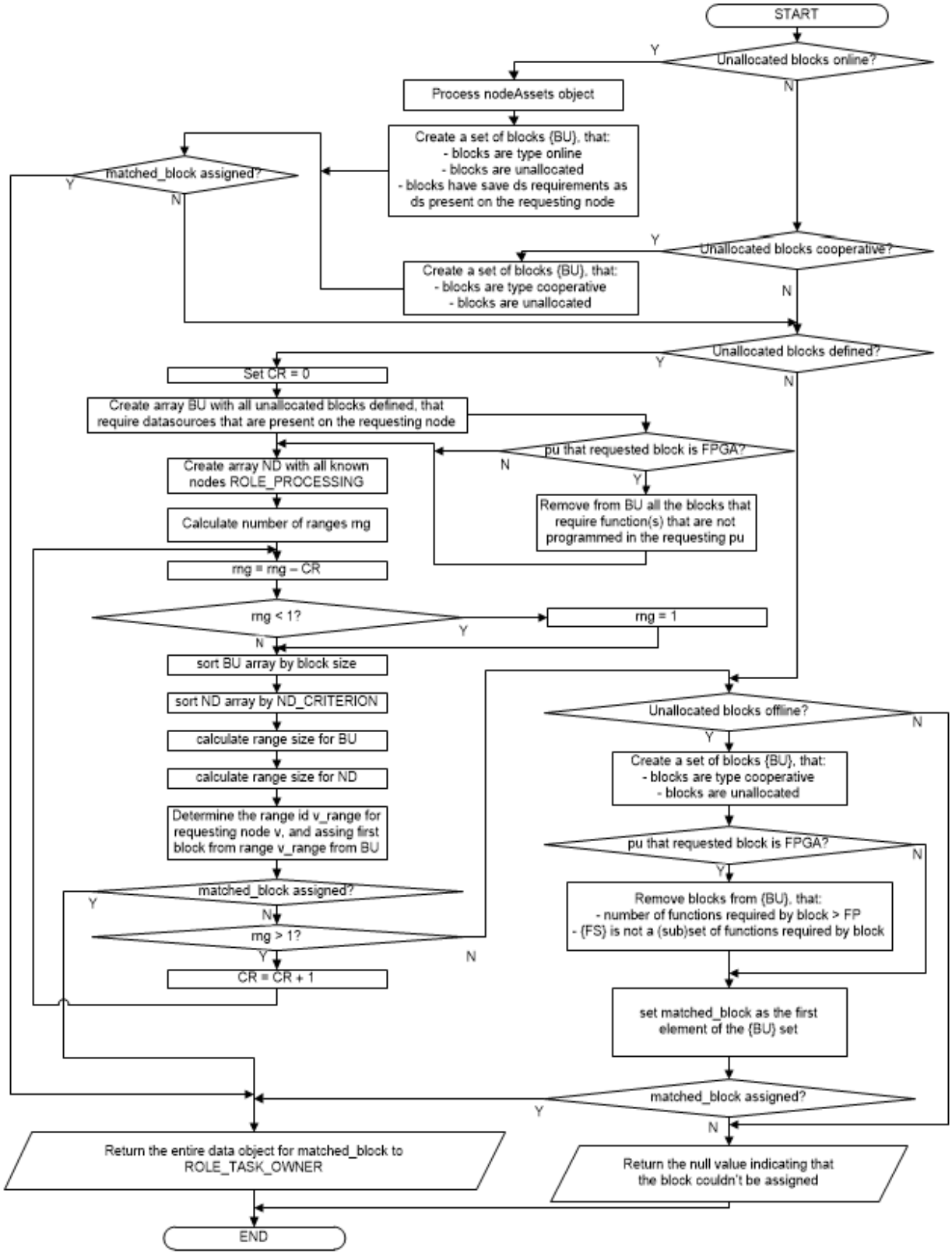


Fig. 41. Operation of AL_MATCH_BLOCK_TO_PU_4 algorithm

AL_MATCH_BLOCK_TO_PU_5

Algorithm presented in this subsection is a benchmark algorithm – that allocates blocks on random basis. The purpose of this algorithm is to evaluate how different are the results from algorithms presented in earlier subsections from the results delivered by simple random-based solution. The operation is shown in Fig. 42.

Algorithm: AL_MATCH_BLOCK_TO_PU_5

```
1. if online blocks available
     Analyze nodeAssets
     Set {BU} -> unallocated blocks online, that:
     Datasources required by block = datasources present on node
2. else if cooperative blocks available
     Set {BU} -> unallocated blocks cooperative
3. if {BU} is not empty
     matched_block <- {BU}.first
     return matched_block
4. if offline blocks available
     BU -> offline blocks that are unallocated
5. if defined blocks available
     add defined blocks that are unallocated to {BU}
6. matched_block <- random element from {BU}
7. if matched_block assigned
     return matched_block
   else return null
```

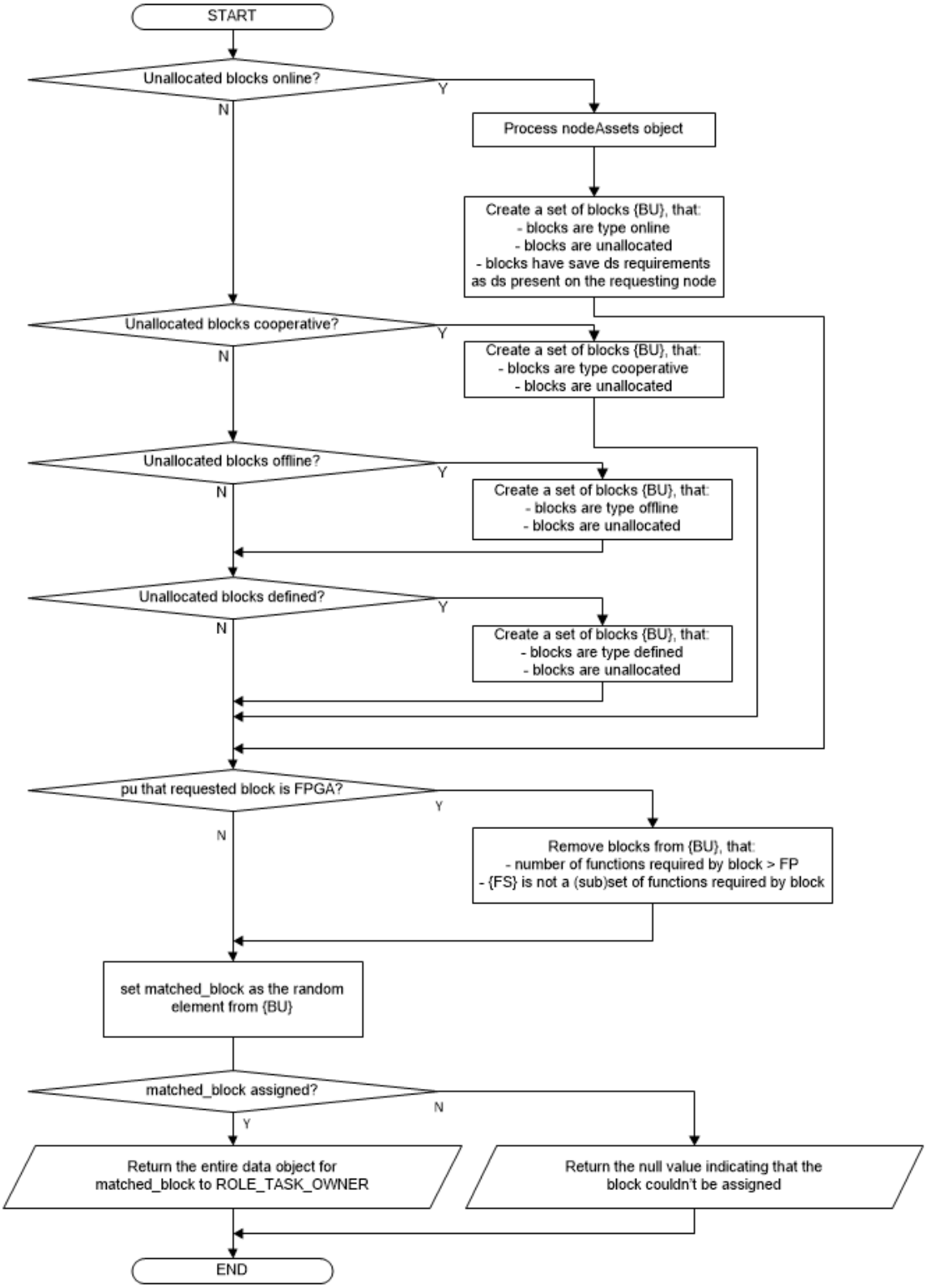


Fig. 42. Operation of AL_MATCH_BLOCK_TO_PU_5 algorithm

4.5.3. Other algorithms

AL_DESIGNATE_CONTROL_ROLES

This algorithm is used to assign roles of ROLE_CONTROL during the DP_PHASE_INIT stage. Here, a simple mechanism is used by selecting the node with the highest transmission speeds.

AL_REASSIGN_ROLES

Algorithm that is executed after task finishes processing. The design of DPRS system allows the AL_REASSIGN_ROLES algorithm to be executed during task processing as well. AL_REASSIGN_ROLES can call the AL_DESIGNATE_CONTROL_ROLES algorithm to reassign the ROLE_CONTROL, or can use a separate mechanism for reassignment.

AL_ALLOW_RECONF

This is the indicator whether reconfiguration is used (AL_ALLOW_RECONF_2) or not (AL_ALLOW_RECONF_1). If the AL_ALLOW_RECONF_1 is applied, then no *reconf_counter* is used and the RECONFIGURATION_PERIOD is not calculated. The FPGA processing units are pre-programmed with functions in this case.

4.6. Other mechanisms

The weak-node exclusion mechanism

The proposed system is equipped with a mechanism that allows controlling the assignment of blocks to the nodes. It operates independently of any AL_MATCH_BLOCK_TO_PU algorithm, for the sake of extended flexibility. The weak-node exclusion mechanism defines the *rejection_threshold* value. Each processing unit on each node has the *opt_metric* value calculated, based on (2) formula (other formula can be used instead, if other criteria are needed).

$$opt\_metric = \frac{a \cdot v_{up} + b \cdot v_{dn} + c \cdot c_p}{\begin{array}{c} d \cdot (OPEX\_W\_COMM\_HEADER + OPEX\_W\_COMM\_PBT) + \\ e \cdot (OPEX\_W\_BLOCKO\_SLOT + OPEX\_W\_BLOCK\_SLOT) + \\ g \cdot (OPEX\_W\_PU\_PER\_SLOT) \end{array}} \quad (2)$$

During each execution of the procedure of matching block to processing unit, the *opt_metric* of the candidate processing unit is compared with *rejection_threshold*. If the *opt_metric* is smaller, then the processing unit is a candidate for rejection. It is called a candidate, because even for the processing units that have *opt_metric* below *rejection_threshold*, they are granted the block regardless bad *opt_metric* – every *task_rejection_rate* times they would be normally rejected. This prevents the starvation of the processing units. The number of tries, after which processing node receives the block anyway is calculated using (3):

$$task\_rejection\_rate = \frac{\left\lceil \frac{\sum_{b=1}^{B} \Psi_b}{V} \right\rceil}{h} \quad (3)$$

The *a, b, c, d, e, f, g, h* are the tuning coefficients and are introduced into (2) and (3) to provide more flexibility. However, for all the research experiments taken further, the default values were used: $a = b = c = d = e = f = g = 1$, $h = 10$, along with the *rejection_threshold* = 0.1.

Step function

The proposed function *H'* is ae variant of the Heaviside function in its discrete form. For the case under consideration, Heaviside discrete function is not suitable, as the step has strict inequality at the left side, while the work presented requires strict inequality at the right side. Therefore, the proposed step function *H'* has the following discrete form:

$$H'(n) = \begin{cases} 0 & n \leq 0 \\ 1 & n > 0 \end{cases} \quad (4)$$

CHAPTER 5

EXPERIMENTAL STUDY AND ANALYSIS

5.1. The experimentation plan

In this chapter, the evaluation of the proposed solutions is included. Due to the novelty of the proposed ideas, no benchmark algorithms exist for comparison. The operational algorithms are compared with simple ones, showing the advantage of using the evaluated algorithms. The following algorithms are to be evaluated:

For nodes of role ROLE_PROCESSING:

- AL_ALLOW_RECONF: impact of using reconfiguration
- AL_RECONFIGURE_FPGA: determine the reconfiguration of node's reconfigurable units

For nodes of ROLE_TASK_OWNER:

- AL_ASSIGN_BLOCK_TO_PU: assigning block to the processing unit while replying to MSG_REQUEST_BLOCK

The following properties are to be evaluated:

- PR_LOCAL_COMP: enabling the node of ROLE_TASK_OWNER to process blocks belonging to his own task(s)
- RECONFIGURATION_PERIOD: the length of period of considering reconfiguration

The following metrics are used to evaluate the quality of the proposed solutions:

Energy utilization

The energy utilization comprises the following elements, listed in Tab. 2:

Table 2. Energy expenditure elements

Element	Description
OPEX_C_ADDING_CONTROL	Cost of adding control nodes. Includes all the operations executed in order to obtain and process the information about control nodes. Does not include the transmission / communication costs.
OPEX_T_GETTING_TASKS	Cost of obtaining task(s) information. Includes all the operations executed in order to obtain and process the information about active tasks. Does not include the transmission / communication costs.
OPEX_G_GETTING_BLOCK	Cost of getting a block from ROLE_TASK_OWNER. Includes all the operations executed in order to get the block for processing, including the transmission costs.
OPEX_D_GETTING_DS	Cost of getting a datasource value from a node that has this datasource installed. Includes all the operations executed in order to get the block for processing, including the node selection and transmission costs.
OPEX_F_GETTING_FN	Cost of getting a function from the ROLE_TASK_OWNER. Includes all the operations executed in order to get the block for processing, including transmission costs.
OPEX_P_PROCESSING_BLOCK	Cost of processing a block defined or block offline that is already present on the node. Includes computation costs.
OPEX_U_UPLOADING_RESULT	Cost of transmitting the result from ROLE_PROCESSING to ROLE_TASK_OWNER. Concerns the results of the processing of blocks defined and blocks offline.
OPEX_O_PROCESSING_BLOCK	Cost of processing a block online or block cooperative that is already present on the node. Includes computation costs and sending out the results.
OPEX_R_FPGA_DECIDE_RECONF	Cost of executing the reconfiguration algorithms. Occur on ROLE_RECONFIGURABLE nodes, separately for each processing unit. Does not include downloading bitstream and programming the bitstream.
OPEX_W_FPGA_DOWNLOAD_BITSTREAM	Cost of downloading the bitstream.
OPEX_I_FPGA_PROGRAMMING_BITSTREAM	Cost of programming the bitstream into processing unit.
OPEX_W_PU_PER_SLOT	Cost of the operation of the processing unit, includes the constant cost of a processing unit being powered on.

The total energy expenditure for the system is the sum of all elements occurring on nodes (5).

$$E = \sum_{v}^{V} \sum_{t}^{T} \sum_{p}^{P} \left(v^{t}_{\text{OPEX\_C\_ADDING\_CONTROL}} + v^{t}_{\text{OPEX\_T\_GETTING\_TASKS}} + v^{t}_{\text{OPEX\_G\_GETTING\_BLOCK}} + v^{t}_{\text{OPEX\_D\_GETTING\_DS}} + v^{t}_{\text{OPEX\_F\_GETTING\_FN}} + p^{t}_{\text{OPEX\_P\_PROCESSING\_BLOCK}} + v^{t}_{\text{OPEX\_U\_UPLOADING\_RESULT}} + p^{t}_{\text{OPEX\_R\_FPGA\_DECIDE\_RECONF}} + v^{t}_{\text{OPEX\_W\_FPGA\_DOWNLOAD\_BITSTREAM}} + p^{t}_{\text{OPEX\_I\_FPGA\_PROGRAMMING\_BITSTREAM}} \right) + Tp_{\text{OPEX\_W\_PU\_PER\_SLOT}} \quad (5)$$

obj^{t}_{elem} indicates the energy that object *obj* spends on operations executing element *elem* during time slot *t*. Each element from Tab. 2 indicates the energy expenditure per slot.

E is constituting the overall summary cost of the system operation. However, each element from Tab. 2 can be defined separately. This provides the flexibility of modeling the energy utilization for various systems.

For some of the experiments, the separate observation of processing and transmission costs is important. Thus the processing cost E_p and transmission cost E_t are defined as follows:

$$E_t = \sum_{v}^{V} \sum_{t}^{T} \sum_{p}^{P} \left(v^{t}_{\text{OPEX\_C\_ADDING\_CONTROL}} + v^{t}_{\text{OPEX\_T\_GETTING\_TASKS}} + v^{t}_{\text{OPEX\_G\_GETTING\_BLOCK}} + v^{t}_{\text{OPEX\_D\_GETTING\_DS}} + v^{t}_{\text{OPEX\_F\_GETTING\_FN}} + v^{t}_{\text{OPEX\_U\_UPLOADING\_RESULT}} + v^{t}_{\text{OPEX\_W\_FPGA\_DOWNLOAD\_BITSTREAM}} \right) \quad (6)$$

$$E_p = \sum_p \sum_t^T \left(p^t_{\text{OPEX\_P\_PROCESSING\_BLOCK}} + p^t_{\text{OPEX\_R\_FPGA\_DECIDE\_RECONF}} + p^t_{\text{OPEX\_I\_FPGA\_PROGRAMMING\_BITSTREAM}} \right) + Tp_{\text{OPEX\_W\_PU\_PER\_SLOT}} \quad (7)$$

Energy model used for experimentation process

The definitions used for the experimentation presented in section 5.4, are as follows. Both the system architecture, (5) definition and the experimentation system allow any changes in the energy elements definitions.

The system is described with the parameters presented in Tab. 3. All the parameters are specified in the input simulation file.

Table 3. Energy modeling parameters

Parameter	Description	Hardware Assignment
OPEX_W_FPGA_PROGRAMMING_HEADER	Constant cost of programming the FPGA, independent of the bitstream size	processing unit
OPEX_W_FPGA_PROGRAMING_PBT	Cost of programming FPGA per kB	
OPEX_W_PBT	cost of processing block per kB	
OPEX_W_DS	cost of processing datasource	node
OPEX_W_COMM_PBT	cost of communication, per kB	
OPEX_W_BLOCK_SLOT	cost of block processing per slot	
OPEX_W_BLOCKO_SLOT	cost of block processing per slot, for blocks online and blocks cooperative	

Hardware assignment – to which hardware unit the parameter cost is tied to. Each parameter is defined individually for each processing unit or node. The scheme used in experiments for 5.4 provides the clear evaluation of each of the energy elements and is defined as follows:

OPEX_C_ADDING_CONTROL = OPEX_W_COMM_PBT

OPEX_T_GETTING_TASKS = OPEX_W_COMM_PBT

OPEX_G_GETTING_BLOCK = OPEX_W_COMM_PBT $\cdot \frac{v_{dn}}{OPEX\_G\_COEFF}$

OPEX_D_GETTING_DS = OPEX_W_DS

OPEX_F_GETTING_FN = OPEX_W_COMM_PBT

OPEX_P_PROCESSING_BLOCK = OPEX_W_BLOCK_SLOT

OPEX_U_UPLOADING_RESULT = OPEX_W_COMM_PBT $\cdot \frac{v_{up}}{OPEX\_G\_COEFF}$

OPEX_O_PROCESSING_BLOCK = OPEX_W_BLOCKO_SLOT

OPEX_R_FPGA_DECIDE_RECONF = OPEX_W_FPGA_PROGRAMMING_HEADER

OPEX_W_FPGA_DOWNLOAD_BITSTREAM = OPEX_W_COMM_PBT$\cdot v_{dn}$

OPEX_I_FPGA_PROGRAMMING_BITSTREAM = $\cdot \frac{\text{OPEX\_W\_FPGA\_PROGRAMMING\_PBT}}{c_p}$

OPEX_W_PU_PER_SLOT = OPEX_W_PU_PER_SLOT

OPEX_G_COEFF is the coefficient used in the simulation process to scale the transmission speeds.

The numerical values used for the simulation have been chosen according to [SCP02], [KDW10] and [BTL10]. Study shows that FPGAs are more energy efficient than CPUs, the FPGA/CPU ratio strongly depends on the type of application and the ability of parallelism. The cautious ratio was chosen for simulations in this work (4:1 in favor of FPGA), while study shows that the benefit of

using FPGA could be much higher. The remaining energy parameters were selected to realistically reflect the real electronic hardware operation.

Task execution time

One of the efficiency factors is the number of slots required to process the task *z*. This time is measured by the experimentation system and evaluated in section 5.4.

Utilization rates

Having the system resources available, they should be utilized for most of the time. Having them in idle state means inefficiency of the system. The utilization of the processing units during the task execution time is evaluated in section 5.4. as one of the efficiency factors.

5.2. The evaluation environment

In order to perform the experiments, a distributed processing simulator was created, solely for the purpose of the research presented here. No known simulator was able to handle the object-based structure including: nodes performing the computations; reconfiguration of node's reconfigurable chips during the runtime; messaging communication implementing encapsulation of data objects; concurrent execution of functions on nodes; executing the functions of nodes in a continuous manner, constituting the logical datasource; dynamic assignment of the roles to the participants of the system.

The experimentation environment was built in a way that fully reflects the distributed and object-based nature of the proposed system. Each entity listed in Chapter 4 is represented as an autonomous object. Node objects run their operational algorithms and communicate with the other nodes in the system. The network layer is fully simulated, reflecting the transmission speeds of all network interfaces, and the sizes of all the messages sent in the system. The networking layer also implements the timeouts and loss of messages – along with all the mechanisms of recovering from such a situation.

The key of the experimentation system is its realtime operation. Unlike many other experimentation systems that calculate the result based on the input data, the experimentation system used for the presented research operates fully online. In this case, *online* term should be considered as each object runs concurrently, and all the procedures (such as data transmission) run along with the nodes operation and independently. The function implementations and bitstream are truly represented as a binary code, sent between nodes and truly executed by nodes for the purpose of block processing. Function implementations are created and compiled with C++, and nodes execute those using threads. Executing functions using FPGA includes the execution wrapper and a binary code representing the bitstream. Another aspect of the *online* operation is the length of the given operation. The designed experimentation system involves the real processing times, related to the computational efficiency of the processing unit. Hence, a given block *b* will be processed in a shorter time on node *v* than on node *w*, if the computational efficiency of node *v* is higher than on node *w*. Such phenomena is fully reflected by the experimentation system. The transmission speeds are also fully implemented in the networking layer. The transmitting time of the message depends on sender's upload speed, receiver's download speed and the size of the message. System includes the timeline, with the atom time slices defined (slots). This way it is

possible to determine the state of each object during each slot, and hence easy construction of the Gantt graphs.

All the objects in the experimentation system implement the architecture presented in Chapter 4, and implement the algorithms in modular way. Several layers have been used, to separate application, processing, control and transmission. Flexible replacement or addition of new algorithms has been implemented. This way the architecture is ready for the hardware implementation without major modifications, and the evaluation results closely reflect the operation of the hardware-based system.

The operation of the experimentation system has been parametrized to make the experimenting efficient, and all the parameters, selected algorithms and output type can be set from the command line. The researched DPRS structure and the input application is described in the form of input text files that are passed to the simulator through command line. Experimentation system also includes input files generator for the convenient generation of the DPRS structures and application tasks.

The list of possible options for the evaluation system is listed below:

```
DIRECPS INPUT FILES GENERATOR
Usage:
    -o, --name                      name of the file/system
    -m, --mode                      mode: sys, task, pu, fun
    -n, --nodes                     number of nodes
    -c, --cnodes                    number of control nodes
    -p, --pu a,b                    number of pu:s (range)
    -s, --ds a,b                    number of datasources (range)
    -u, --up a,b                    upload speed (range)
    -d, --dn a,b                    download speed (range)
    -a, --opexCH a,b                opex comm header (range)
    -b, --opexCP a,b                opex comm pbt (range)
    -t, --task                      name of task
    -z, --end                       end slot
    -P, --puavail a,b,c...          available pu:s (list)
    -S, --dsavail a,b,c,...         available ds:s (list)
    -k, --taskid                    task id
    -w, --owner                     task owner
    -r, --rawl                      raw data length
    -g, --dsoff                     number of ds required by offline blocks
    -f, --fnoff                     fn required by offline blocks
    -F, --fnavail a,b,c...          available fn:s (list)
    -l, --onlen a,b                 length of online blocks (range)
    -j, --deflen a,b                length of defined blocks (range)
    -e, --fn a,b                    number of functions required by
                                    defined_blocks (range)
    -x, --blon                      number of online blocks
    -y, --blde                      number of defined blocks
    -h, --help                      Prints this help
```

5.3. Energy utilization

This section presents the efficiency evaluation, where the electrical energy usage is used as a metric. Due to the nature of the experimentation system, as it closely reflects the real systems, the experiment results are averaged within the results for the same input. The (5) formula is used to determine the value of the energy spent.

The following way of comparing two values AL1 and AL2 is used:

$$C_{AL2}^{AL1} = \frac{AL2 - AL1}{AL2} \cdot 100\% \tag{8}$$

5.4. Experiments

Based on [BTL10], the energy consumption for FPGAs is significantly less for the same task, compared to CPUs. However, the management, reconfiguration and operational algorithms for the reconfigurable distributed processing system presented here include additional energy costs. In the process of the research experimentation, the results presented further in this chapter are based on roughly ~4000 simulations (part of them are presented, part were used to confirm the results).

5.4.1. Impact of share of FPGAs

Share of FPGAs: $S = \frac{number\ of\ FPGA\ pu:s}{P} \cdot 100\%$

For three analyzed tasks, increasing the *S* was lowering the operational cost for the DPRS. For all experimented tasks and systems, the *S* threshold S_{thres} has been observed and defined as:

$$S_{thres} \rightarrow S: \forall\ S_{hi} > S,\ |E_{hi} - E_{thres}| < \varepsilon$$

This way, for a given DPRS executing a task, there exists a threshold value S_{thres} of S, for which further increments in S does not reflect in the reduction of E higher than ε. The relation between FPGA pu:s and functions they are capable of programing explains this phenomenon. As the AL_ALLOW_RECONF=AL_ALLOW_RECONF_1 for this experiment, FPGAs do not drop functions already programmed. Therefore, they are being configured to execute a set of functions F_P (and hence, a set of blocks that require only functions from set F_P) and this set is limited. Thus, FPGAs can process the limited number of blocks with specific requirements (determined by AL_RECONFIGURE_FPGA), and as the set of blocks with requirements F_P is limited, then further increase of S does not in effect cause a larger number of blocks to be processed by FPGAs. Therefore, the effective values of S_{thres} need to be determined for DPRS.

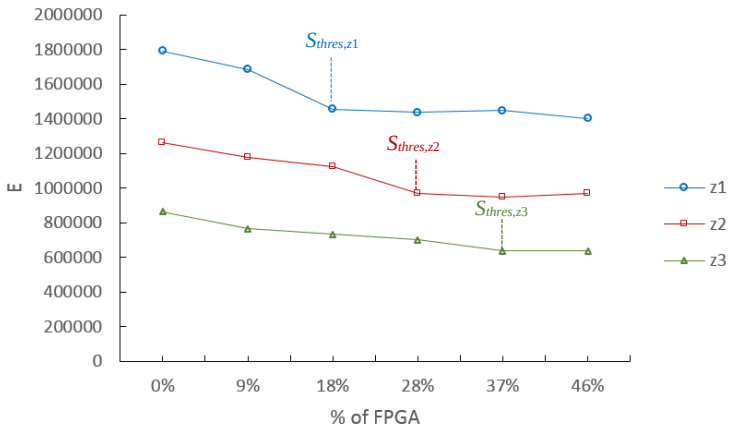


Fig. 43. Cost per % of FPGAs and S_{thres} illustration

As shown in Fig. 43, the values of S_{thres} are the following: ~18% for z_1, ~28% for z_2, and ~37% for z_3. Regarding the total sizes for each task (defined as $\Psi_z = \Psi_{blocks\_offline} + \Psi_{blocks\_online} + \Psi_{blocks\_defined} + \Psi_{blocks\_cooperative}$), they were set to conform the rule:

$$\Psi_{z1} > \Psi_{z2} > \Psi_{z3} \quad (9)$$

The experiment described above, along with multiple other experiments confirm, that for tasks conforming (9) and (10) occurs:

$$S_{thres,z1} < S_{thres,z2} < S_{thres,z3} \quad (10)$$

The phenomenon of S_{thres} has been identified during this research and has been addressed – the results of the proposed solutions are described later in this section.

5.4.2. The impact of AL_RECONFIGURE_FPGA

This experiment shows that the DPRS behavior depends on the AL_RECONFIGURE_FPGA algorithm. For all the experiments, the AL_ALLOW_RECONF=AL_ALLOW_RECONF_2 is set – then the online reconfiguration ability is used to measure the impact of FPGA pu:s reconfiguration on the energy usage and the operation time. The experiments were run for 5 sets of input data having various total size of blocks (23.8MB, 84.7MB, 166MB, 328.5MB and 653.6MB). The design had 27 nodes, 1-4 pu:s, 1-2 datasources, transmission speed varying 5-50kB/slot and 20-80kB/slot for upload and download respectively. Results from 36 simulations and 108 measurements have been used. Two types of pu:s have been used: CPU and FPGA. The presented results and relations are true for other networks also.

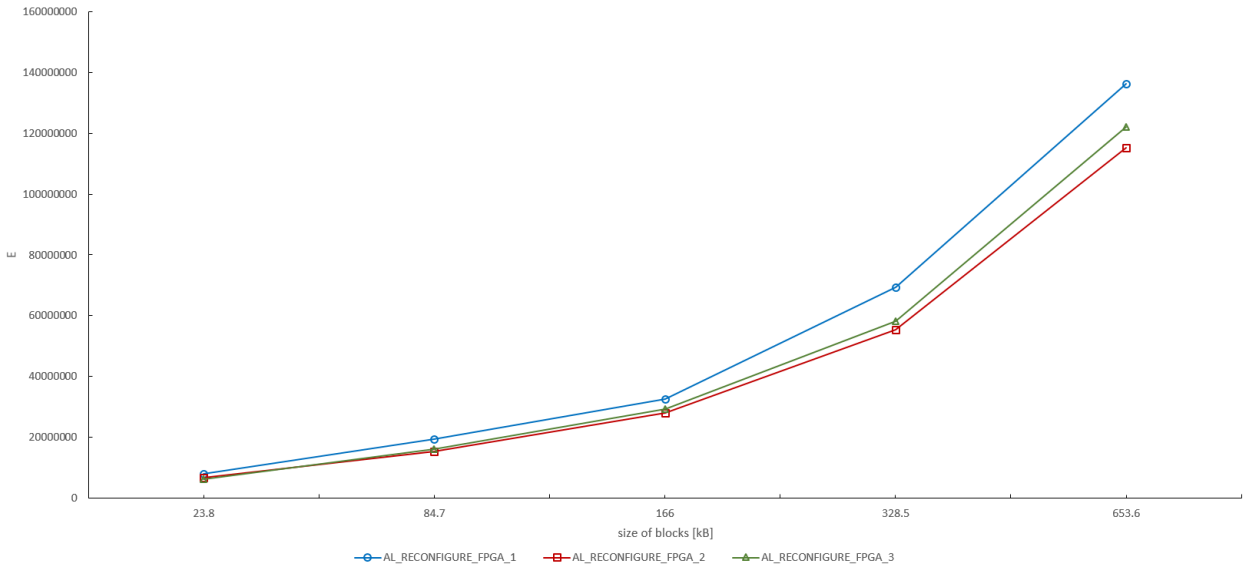


Fig. 44. FPGA reconfiguration algorithms impact to E

The even distribution of the functions selected to be programmed onto FPGA, used in AL_RECONFIGURE_FPGA_1 exposed high operational cost, and the differences compared to the AL_RECONFIGURE_FPGA_2 and AL_RECONFIGURE_FPGA_3 differ depending on the size of processed blocks (Fig. 44, up to 22% calculated using (8)). AL_RECONFIGURE_FPGA_2 uses the information about blocks' requirements and returns f_{cand} being most universal for the current block resources and suitable for most blocks. AL_RECONFIGURE_FPGA_3 works similarly, but also matches functions with costs related with getting datasources values. The results show the importance of using proper reconfiguration algorithm that will adopt to the current state of the system.

In AL_RECONFIGURE_FPGA_3 the Φ_f calculation includes only those blocks that do not require datasources that are non-local for the node requesting the reconfiguration metrics. For AL_RECONFIGURE_FPGA_2, all blocks (except *online* blocks) are taken in to consideration.

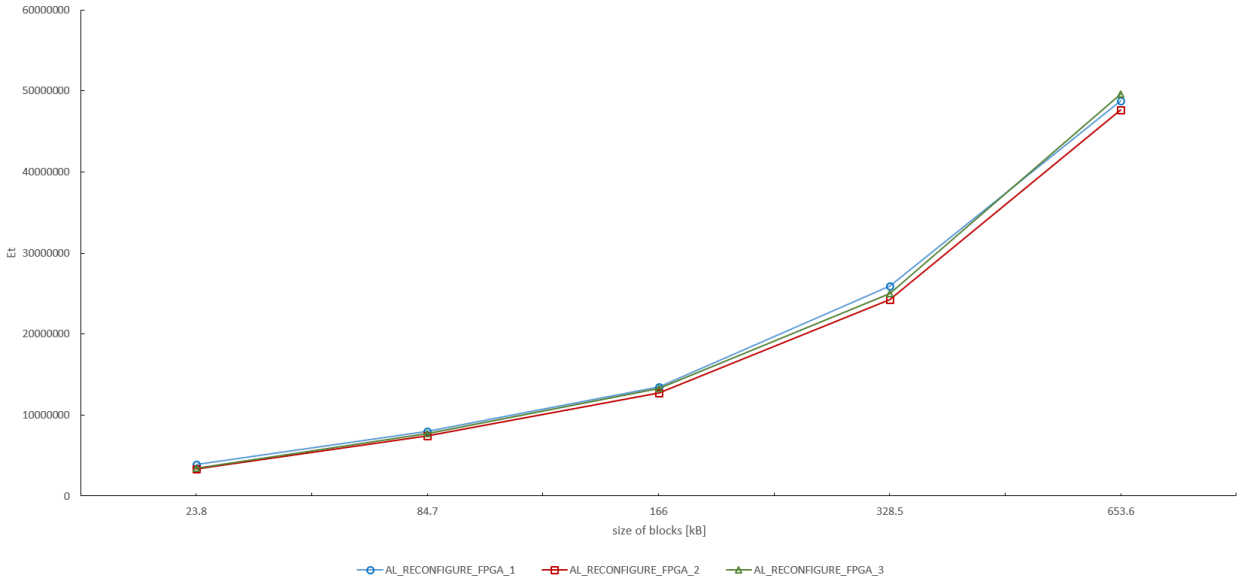


Fig. 45. Communication costs E_t

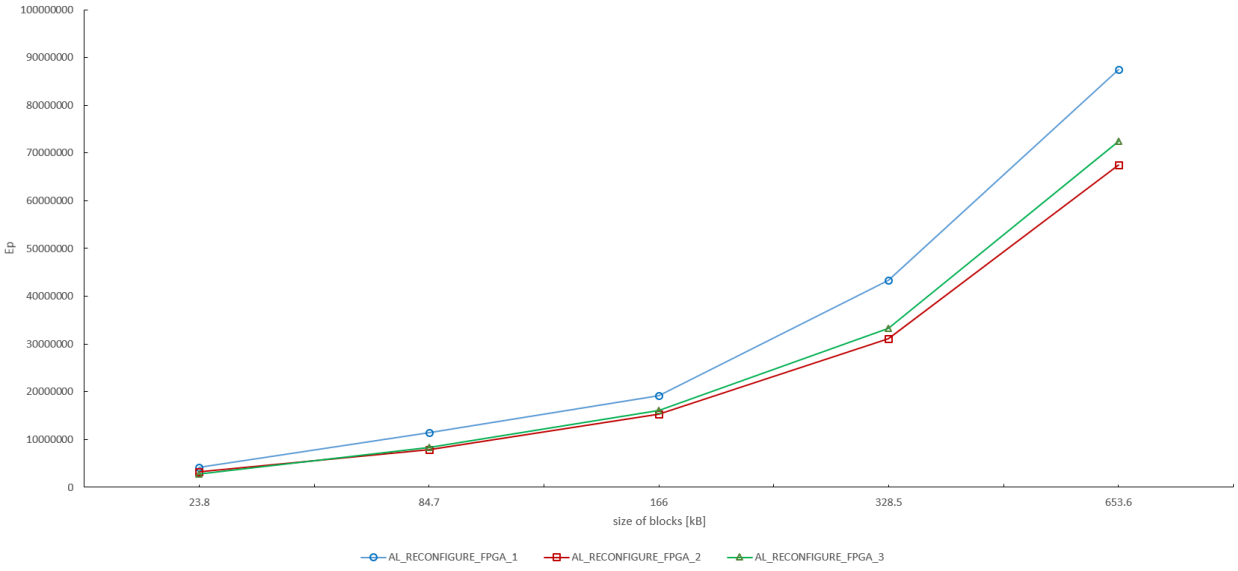


Fig. 46. Computation costs Ep

To show the profit and properties of AL_RECONFIGURE_FPGA algorithm alone, the communication costs for all nodes are set to the same values (to sort out the cost differences produced just by different nodes' communication parameters). This way, Fig. 44, Fig. 45 and

Fig. 46 show only the part of the cost that is optimized by AL_RECONFIGURE_FPGA. Communication costs are optimized by other algorithms described further. As shown in Fig. 45, communication costs are very similar for all three algorithms – because no matter which node receives a block, the communication cost is always the same (OPEX_W_COMM_PBT is set to equal for all nodes). The main profit of the AL_RECONFIGURE_FPGA algorithm is in optimizing the processing cost, by periodically reprogramming the FPGA with the most suitable function at the given time.

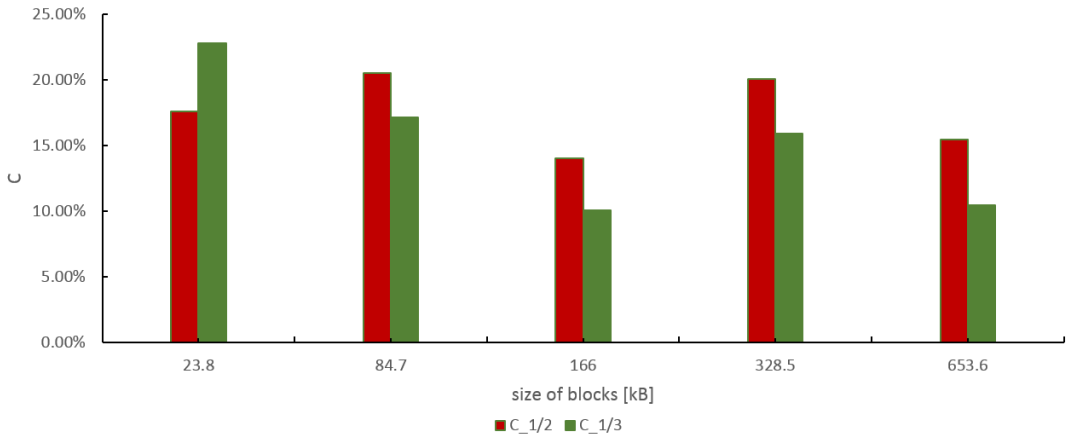


Fig. 47. Values of *C* for AL_RECONFIGURE_FPGA algorithms

Figure Fig. 47 shows the $C_{AL\_RECONFIGURE\_FPGA\_2}^{AL\_RECONFIGURE\_FPGA\_1}$ and $C_{AL\_RECONFIGURE\_FPGA\_3}^{AL\_RECONFIGURE\_FPGA\_1}$. For smaller networks the AL_RECONFIGURE_FPGA_3 performed better than AL_RECONFIGURE_FPGA_2 compared to AL_RECONFIGURE_FPGA_1 (22.78% and 17.62% respectively). For bigger networks, AL_RECONFIGURE_FPGA_2 performed better and is the right choice of algorithm for such networks. The advantage of AL_RECONFIGURE_FPGA_2 comes from the periodic reconfiguration not limited by datasources matching (like it happens in AL_RECONFIGURE_FPGA_3). Thus,

AL_RECONFIGURE_FPGA_2 is much more flexible and adapts better to the current processing needs of the system.

The datasources matching used by AL_RECONFIGURE_FPGA_3 gives the advantage of less reconfigurations while still maintaining the satisfactory cost saving compared to AL_RECONFIGURE_FPGA_1 (more than 10% for all the researched cases). The tradeoff for less reconfigurations, compared to AL_RECONFIGURE_FPGA_2, is always between 4%-5% – this difference is the expense of less reconfigurations.

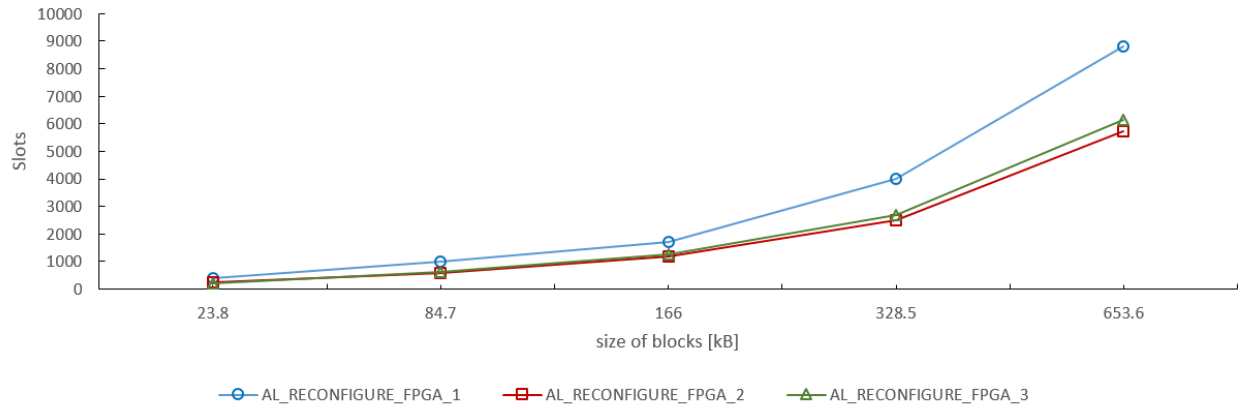


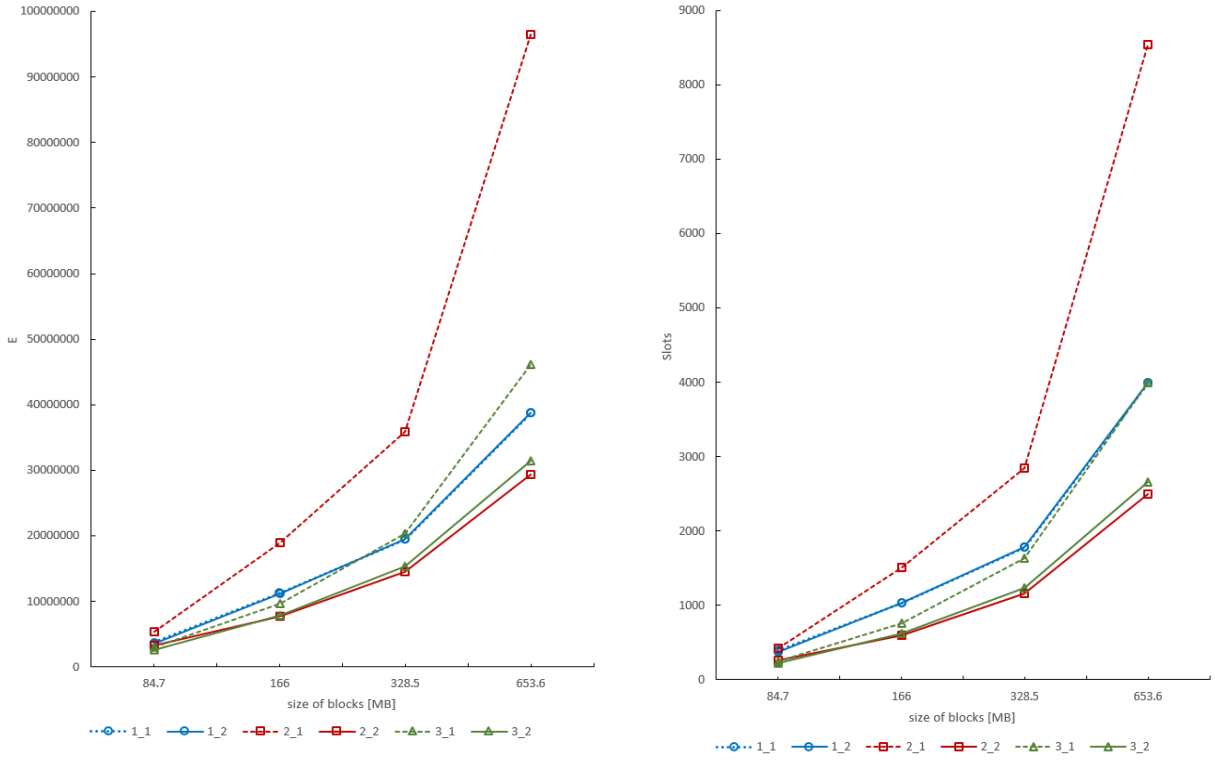
Fig. 48. Processing time required for AL_RECONFIGURE_FPGA algorithms

The selection of AL_RECONFIGURE_FPGA also impacts the processing time. AL_RECONFIGURE_FPGA_1 required the longest time of processing, and the difference compared to the remaining two algorithms increased with the increase of the task(s) size. AL_RECONFIGURE_FPGA_2 is 31%-38% faster, and AL_RECONFIGURE_FPGA_3 is 27%-44% faster – compared to AL_RECONFIGURE_FPGA_1 algorithm (Fig. 48).

Concluding, the AL_RECONFIGURE_FPGA_3 should be used in cases when it is important to minimize the number of reconfigurations and when fetching the remote datasources is not preferred. AL_RECONFIGURE_FPGA_2 should be used in all other situations.

5.4.3. Enabling the reconfiguration with AL_ALLOW_RECONFIGURE

The dynamic reconfiguration of the FPGAs is a very important aspect for system efficiency. In this section, the reconfigurable architecture is compared with non-configurable one.



a) *E* cost

b) Number of slots required

Fig. 49. Reconfiguration on/off

Fig. 49. shows the impact of turning the reconfiguration off (24 simulations). Dashed lines indicate cases, where reconfiguration is off, while solid lines indicate the reconfiguration turned on. Legend

description *A_B* indicates: *A* → AL_RECONFIGURE_FPGA_*A*, *B* = 1 when reconfiguration is off, *B* = 2 when reconfiguration is on. For AL_RECONFIGURE_FPGA_1, there is no difference in the *E* (pointed line covers solid line). This occurs, because AL_RECONFIGURE_FPGA_1 does not adapt to the current situation in the system – thus the functions programmed onto the nodes resemble the initial configuration for most of the time. AL_RECONFIGURE_FPGA_2, as the algorithm designed for reconfiguration, performs worst when reconfiguration is turned off. It is clearly visible, that FPGAs configuration programmed at the beginning (i.e. using the state of the system in its early state of processing) becomes obsolete while system processes the blocks. This non-matching configuration is especially inefficient in cases, where large amounts of data is processed, resulting in more blocks being processed by inefficiently configured system, that results further in higher electrical energy required to process the task (as high as 70% more energy required). AL_RECONFIGURE_FPGA_3 shows similar properties to AL_RECONFIGURE_FPGA_2, although the differences are smaller (up to 31%). The reconfiguration impacts not only the *E*, but also the time required to process the task(s). The number of slots required, when plotted as a chart, resembles the chart of *E* costs (also for *C* metrics). Conclusion: adjusting the FPGAs configurations during system operation is crucial for minimizing both the electrical energy and the time required to process the task(s).

5.4.4. Reconfiguration period

The architecture of reconfigurable system performs periodic reconfiguration for each reconfigurable processing unit. Such reconfiguration may occur only when a node is not doing the computation at that moment, as the currently programmed function is used for that. Each node is using the same RECONFIGURATION_PERIOD setting that determines the period (measured in

slots *t*) between processing unit's reconfiguration attempts. For a particular node *v*, its processing units will be attempting reconfiguration at different slots, as they will be finishing the processing at various times. Each reconfiguration attempt incurs processing cost, even if reconfiguration is actually not performed (non-reconfiguring attempt may occur for example when the reconfiguration algorithm yields the function that is already programmed). The RECONFIGURATION_PERIOD also determines, how well the system will adjust the current system needs. Rare reconfigurations (i.e. high value of RECONFIGURATION_PERIOD) lead to the situation where FPGAs are programmed with other functions that are currently required by the on-going processing. On the other hand, often reconfiguration attempts cause energy loss for senseless attempts. In such case, reconfiguration attempts are invoked more often than the system needs change. Also, nodes spend a lot of processing resources for reconfigurations, what lowers their overall computing efficiency – leading to longer operations and therefore to higher *E* cost for the particular task(s). Thus, the following hypothesis can be formulated:

For each system configuration – considering nodes' parameters and task(s) – there exist the optimal value(s) of RECONFIGURATION_PERIOD*, for which the E is smallest.*

The above hypothesis was confirmed by numerous experiments for multiple systems of various network and task configurations. Four of the selected result collections are shown in Fig. 50 – Fig. 53. A total of 210 simulations have been performed. Reconfiguration algorithm used: AL_RECONFIGURE_FPGA_2.

Table 4. Configurations used for Fig. 50 – Fig. 53

	Nodes	**task data size**
Configuration 1	27	24 MB
Configuration 2	30	85 MB
Configuration 3	40	170 MB
Configuration 4	45	330 MB

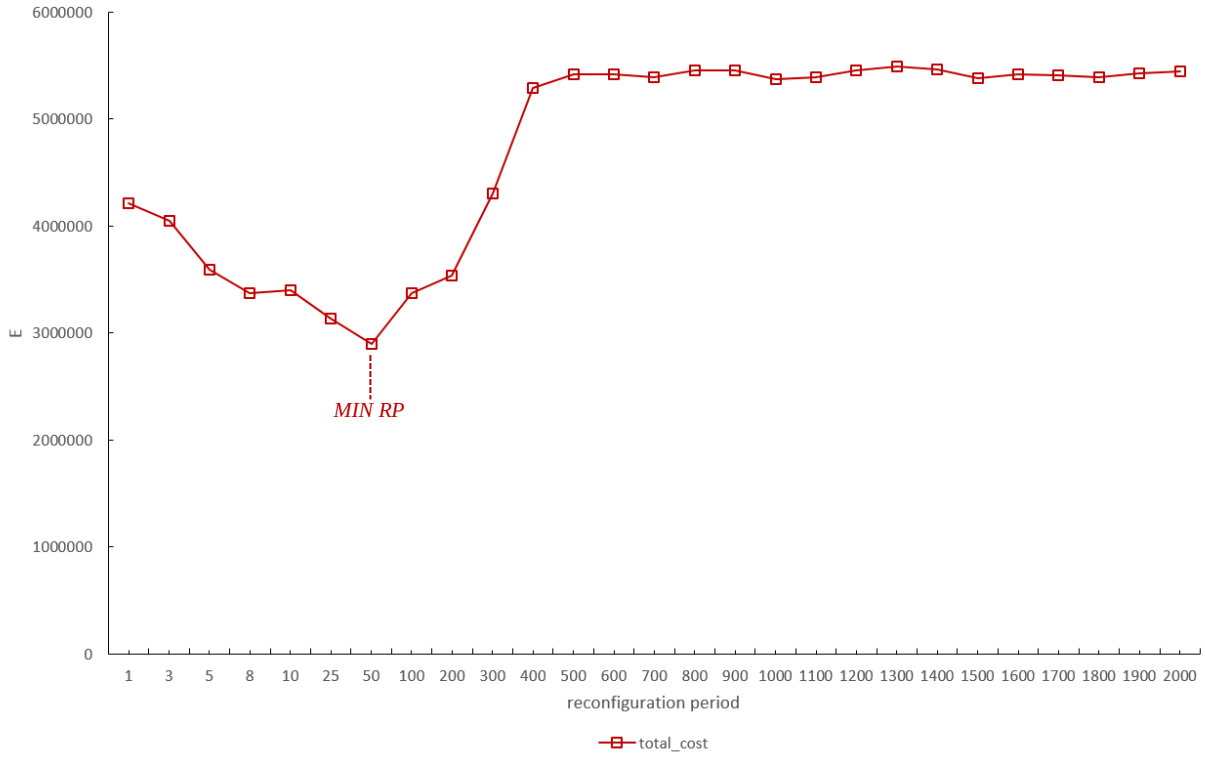


Fig. 50. Total cost as a function of reconfiguration period (configuration 1)

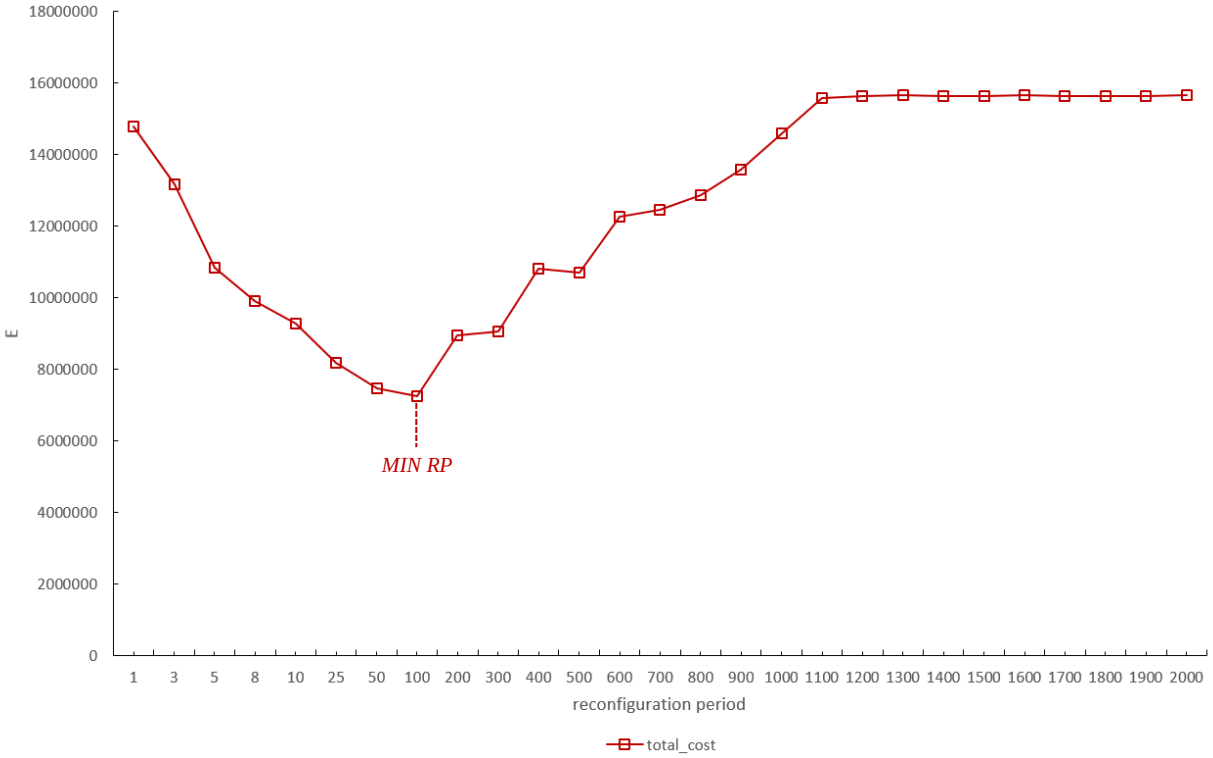


Fig. 51. Total cost as a function of reconfiguration period (configuration 2)

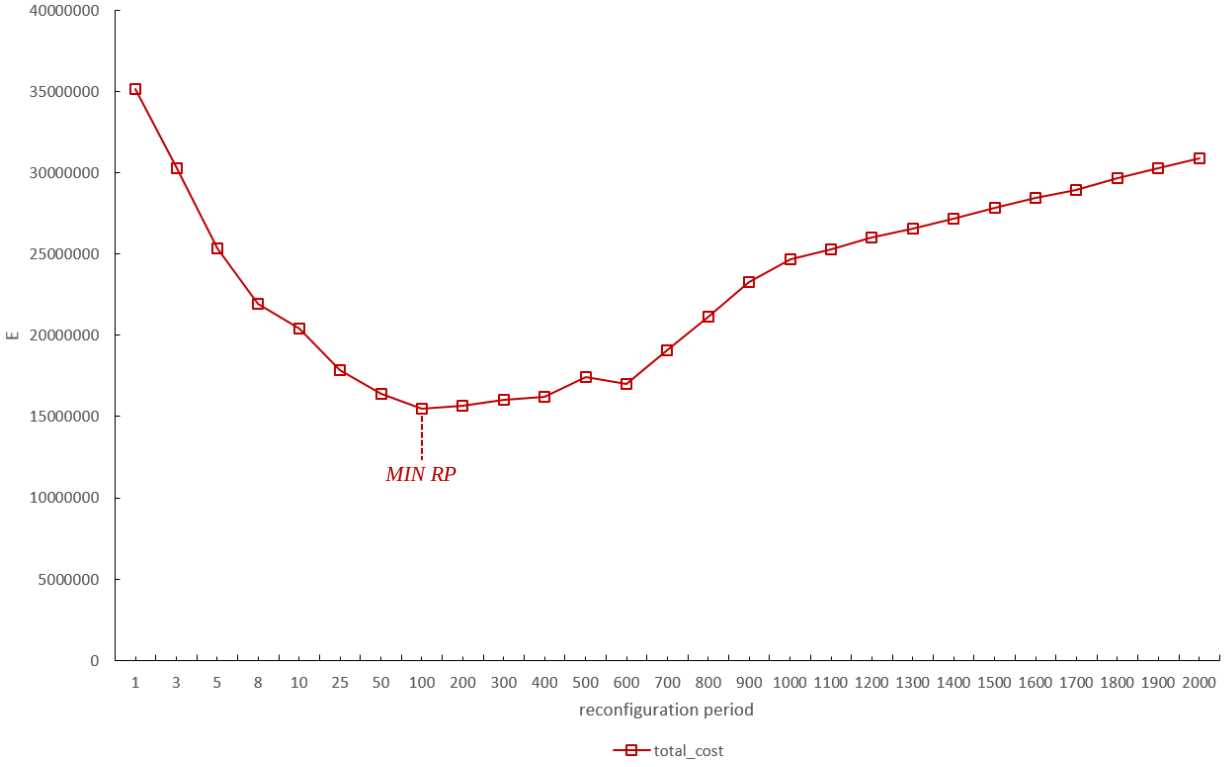


Fig. 52. Total cost as a function of reconfiguration period (configuration 3)

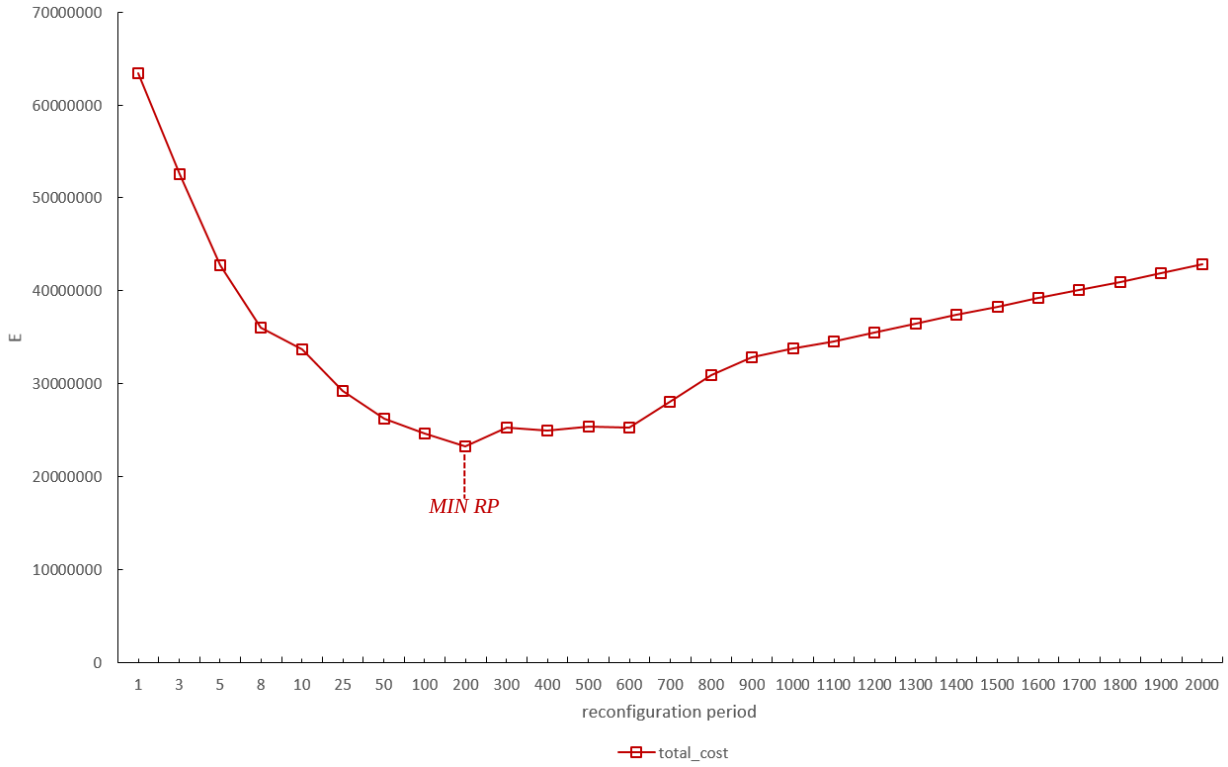


Fig. 53. Total cost as a function of reconfiguration period (configuration 4)

Figures Fig. 50 – Fig. 53 reflect numerous experiments executed over numerous systems and tasks. The configurations used are shown in Tab. 4. It can be observed, that for low values of RECONFIGURATION_PERIOD the *E* is very high, due to massive reconfiguration attempts. Increasing the RECONFIGURATION_PERIOD lowers the *E*, to usually reach the minimum monotonically. This minimum is called θ_Ω (where Ω denotes the DPRS system). Further increase may keep the *E* in values similar to θ_Ω, the range of RECONFIGURATION_PERIOD for which *E* stays moderately close to the minimum differs among system/tasks, but the research shows that it tends to shorten for small task sizes. It was also observed, that often the local minimum $\theta_{\Omega M}$ occurs, usually shortly before *E* significantly rises up. The third stage of the RECONFIGURATION_PERIOD range is the rapid increase of *E*. In this range, the system configuration matches lesser and lesser to the required processing needs. The rarer the

reconfiguration is, the worse match between required and present configuration. Fourth and last stage is the moderately constant *E*, regardless of the RECONFIGURATION_PERIOD change. That's because the time between reconfigurations is longer than the task execution time – therefore the (re)configuration occurs once at the beginning of the system operation. Therefore, the algorithm AL_RECONFIGURE_FPGA_2 acts the same way as AL_RECONFIGURE_FPGA_1.

To address the setting of the RECONFIGURATION_PERIOD, the following formula (11) is proposed to calculate the RECONFIGURATION_PERIOD (*RP*) value before the system operation starts:

$$RP = \max_{f} \left(\frac{M_f}{\sum_{p=1}^{P} c_p} + \frac{M_f}{\sum_{v=1}^{V} v_{dn}} + \frac{M_f}{\sum_{v=1}^{V} v_{up}} + \frac{\left\lceil \frac{SIM\_DS\_GET}{\left\lfloor \frac{\sum_{v=1}^{V} v_{dn}}{V} \right\rfloor} \right\rceil \sum_{b=1}^{B} \lambda_{b,f} H' \left(\sum_{d=1}^{D} l_{b,d} - \left\lfloor \frac{D}{V} \right\rfloor \right)}{\sum_{v=1}^{V} H' \left(\sum_{g=1}^{G} \sum_{p=1}^{P} w_{v,g} u_{p,g} m_{p,FPGA} \right)} \right) \quad (11)$$

The RECONFIGURATION_PERIOD metric estimates its value as the sum of the following components: estimated computation slots, estimated download slots, estimated upload slots and estimated datasources handling slots. Such sum is calculated for each function *f* present in the system, and the maximum (sum value over *f* is selected as the *RP* value used for the system run.

The value M_f states the size (in kB) of the block data that is related to function *f*, i.e. the sum of data sizes of all blocks *defined* and *offline* that require the function *f*:

$$M_f = \sum_{b=1}^{B} \left(b_{t4} \Psi_b n_{b,f} \lambda_{b,f} + b_{t1} \Psi_b n_{b,f} \lambda_{b,f} \right) \tag{12}$$

Ψ_b represents the size of the block *b*, and function $\lambda_{b,f}$ represents the threshold of functions required by block *b*. $\lambda_{b,f}$ is defined the following way:

$$\lambda_{b,F'} = \begin{cases} 1 & 1 \; when \; n_{b,f'} = 1 \; for \; \forall f' \in F' \wedge n_{b,f'} = 0 \; \forall f' \notin F' \\ 0 & otherwise \end{cases} \tag{13}$$

Function $\lambda_{b,F'}$ returns 1 if a block *b* requires just the functions from set *F'*, and doesn't require all the other functions. For the set *F'* containing a single function *f*, it is denoted by the identifier of function *f*. M_f divided by the total computation power gives the estimate of the number of slots required to perform the computation of all blocks requiring function *f*. Analogously, the M_f over download speed and upload speed estimates the number of slots required for blocks related to *f* download and upload, respectively. The estimation of the time required to process the datasources for a given function *f* is done the following way. The average number of datasources per node is computed. Then for each block, that requires more datasources than the average (and also requires function *f*), the estimated datasource processing time is added for each datasource above the average. The sum of these datasource processing times, divided by number of nodes having FPGA

pu:s installed constitutes the fourth component of the *RP* formula. SIM_DS_GET is the energy cost factor, defined as the constant minimal energy involved in the single datasource processing.

To measure the accuracy of (11) in reaching the optimal value of RECONFIGURATION_PERIOD, the measurements analogous to those shown in Fig. 50 – Fig. 53 have been performed for 51 systems (*V*=16-60, task size=23MB-668MB), with a total of almost 2000 simulations performed. Therefore the optimal value $RPO_{\Omega\varepsilon}$ (with the tolerance of $\varepsilon = 5$ slots) has been obtained through the brute-force method of executing the system Ω for increasing values of RECONFIGURATION_PERIOD, instead of using the (11) formula. Then, the value of RP obtained from (11) is compared with $RPO_{\Omega\varepsilon}$ using formula (14):

$$RP_{acc} = \frac{RP - RPO_{\Omega\varepsilon}}{RP} \tag{14}$$

$$RP\_E = \left|\frac{E_{RP} - E_{O\Omega\varepsilon}}{E_{RP}}\right| \tag{15}$$

Experiments show that using the (11) formula keeps the *E* below 6%, which means that in the case of analyzed systems, using the (11) instead of brute force causes no bigger than 6% increase in *E* cost. Fig. 54 shows also the negative values for the RP_E: investigated systems operate real time and thus under same conditions the final result might differ by a small value. However, this result is expected and the percentage value of *RP_E* staying around *X* axis demonstrates the high quality of designating *RP* value using (11). For the second important factor of the DPRS operation – the operational timespan – using (11) is increasing the operational timespan by no more than 7%.

Fig. 54 also shows the relation between the values of RP (from (11)) and $RPO_{\Omega\varepsilon}$ obtained by brute force – RP_{acc} determined by (14).

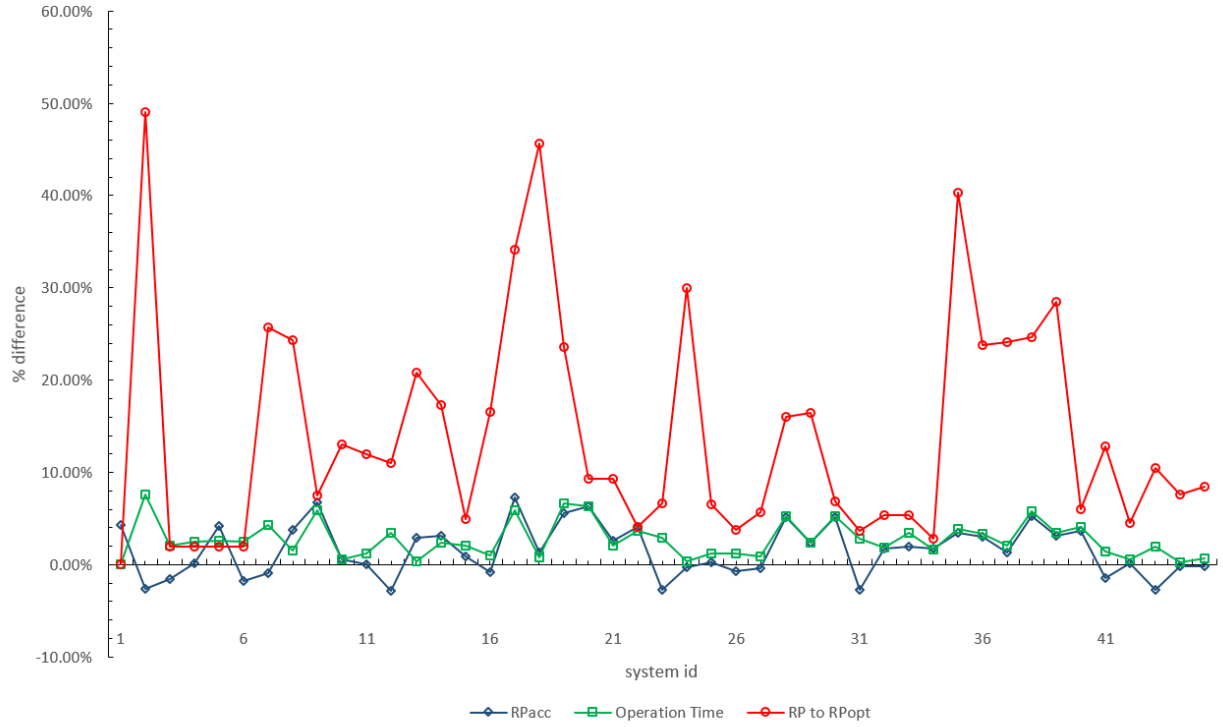


Fig. 54. RECONFIGURATION_PERIOD prediction

In some cases, the high value of RP_{acc} result in a small change of the E or operational time. It is because the relative difference between RP and $RPO_{\Omega\varepsilon}$ is much less important than this relative difference compared to the total system operation timespan. Therefore, the following metric is proposed as more descriptive in this case:

$$RP\_ABS = \frac{RP - RP_{\Omega\varepsilon}}{T_{avg}} \tag{16}$$

Where T_{avg} is the average operational timespan of the system, determined empirically from the experiments. In such case, the *RP_ABS* determines the actual significance of the difference between the brute force best value $RPO_{\Omega\varepsilon}$ and *RP* value obtained from (11).

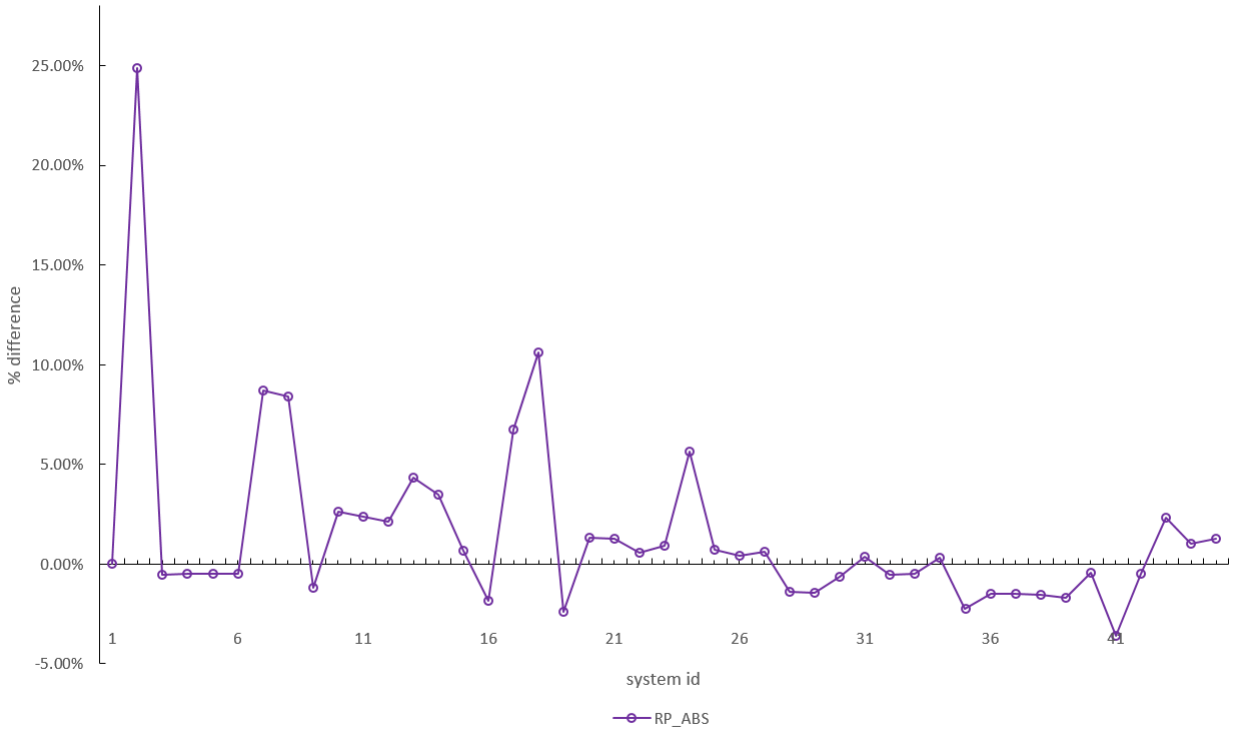


Fig. 55. *RP_ABS* for investigated systems

For the investigated systems, *RP_ABS* stayed mostly below 10%, usually taking the values below 5%, with one peak value of 25% (Fig. 55). This demonstrates that the use of (11) is efficient in most cases.

Also, experiments show that for a value span $RP_{acc} \in \langle RP\text{-}\varepsilon', RP+\varepsilon''\rangle$ the *E* stays in more or less the same areas (ε' and ε'' are different for each system). Thus the *RP* value determined by the

proposed (11) formula should be relatively close to $RPO_{\Omega\varepsilon}$ and stay in the range of $<RP\text{-}\varepsilon', RP+\varepsilon''>$ to keep *E* around efficient level.

5.4.5. Impact of MATCH_BLOCK_TO_PU

This section discusses the way of matching specific blocks to the local resources on the node – to the processing units. As each block requires specific resources, each node possesses certain resources, and each processing unit can have properties too – the block-to-processing unit has the impact on the efficiency and the power consumption of the system. The following algorithms are used: AL_MATCH_BLOCK_TO_PU_1, AL_MATCH_BLOCK_TO_PU_2, AL_MATCH_BLOCK_TO_PU_3, AL_MATCH_BLOCK_TO_PU_4 and AL_MATCH_BLOCK_TO_PU_5 – described in the early part of section 5.3. An important experimentation is done to a set of 11 input datasets modeling the system structure (275 simulations, 825 measurements). Each of them has the same number of units (50 each) but the relation of power-costly nodes to efficient ones *Q* (defined as (17)) varies in the range $Q \in <0.0, 1.0>$. Thanks to this way of investigation, it can be observed what the operation of each algorithm is on sets of efficient and inefficient nodes and how it reflects the electrical efficiency. The classification of a node as *efficient* or *inefficient* is done based on the average processing cost (OPEX_W_PBT, OPEX_W_DS, OPEX_W_BLOCK_SLOT, OPEX_W_BLOCKO_SLOT) for all the processing units installed on a given node.

$$Q = \frac{V - \Phi_{inefficient\_nodes}}{V} \tag{17}$$

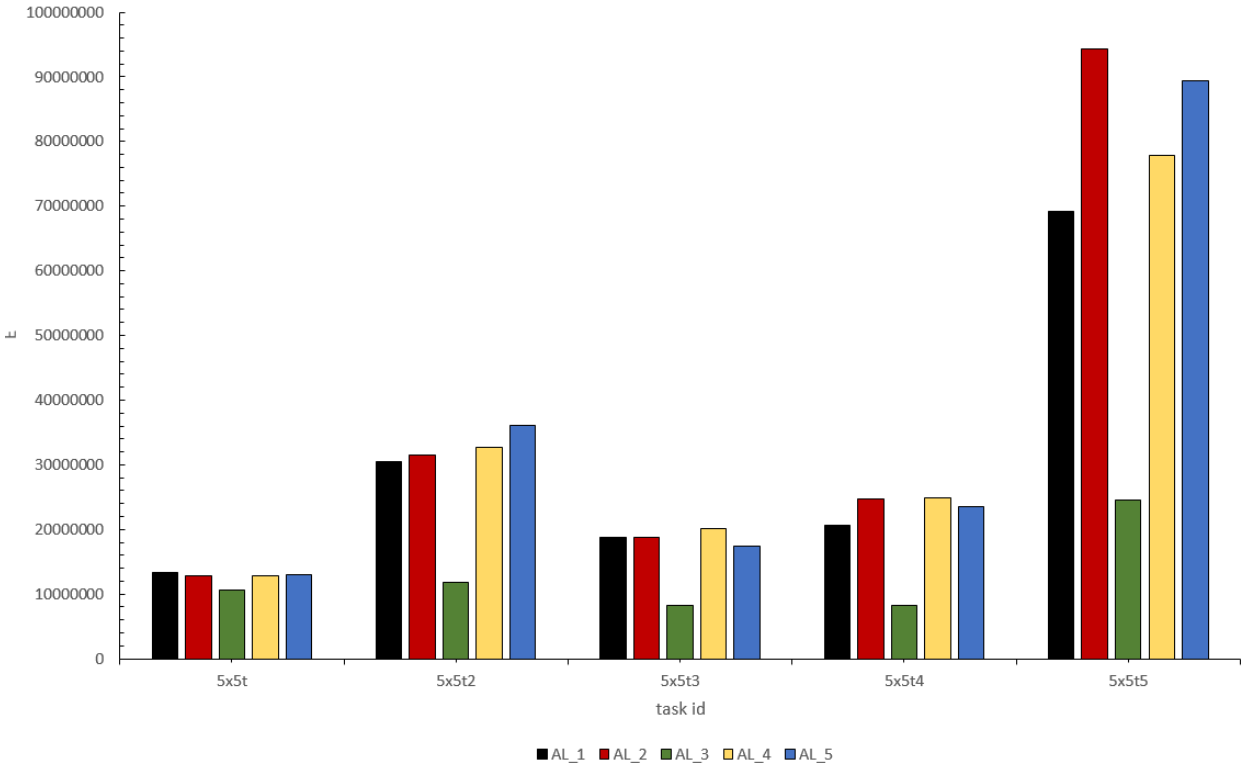


Fig. 56. *E* for *Q* = 0.5

5xt* represent different tasks/applications executed on a given system. Fig. 56 shows the results for the system with *Q* = 0.5 and for 5 different applications. It can be observed that the advantages of different algorithms vary from application to application. Application data for 5x5t, 5x5t2, 5x5t3 and 5x5t4 has been selected in such a way to reflect different scenarios, e.g. high demand for datasources, different number of functions present etc. Application 5x5t6 is an application with high number of large blocks, compared to other applications. Algorithm AL_MATCH_BLOCK_TO_PU_3 demonstrates the highest efficiency in all the cases (also for multiple experiments not presented here). For some of the applications like 5x5t, the difference is not significant, however for applications with a large number of large blocks (5x5t5) the advantage of using AL_MATCH_BLOCK_TO_PU_3 is the greatest. Experiments show that the impact of

the AL_MATCH_BLOCK_TO_PU appears to be the highest in case of systems where many datasources, functions are involved, power consumption of the nodes varies and the requirements of blocks are least uniform. In such cases, the match of block to the processing unit makes a significant change compared to the scenario, where the properties listed above are similar and wrong choice is not changing a lot in the result.

Table 5. Properties of applications used

Application	BLOCK_ONLINE	BLOCK_DEFINED	APPLICATION SIZE
5x5t	small number moderate size	large number small size	143 MB
5x5t2	moderate number moderate size	moderate number moderate size	68 MB
5x5t3	moderate number moderate size	moderate number moderate size	46 MB
5x5t4	moderate number small size	moderate number moderate size	48 MB
5x5t5	moderate number small size	large number large size	150 MB

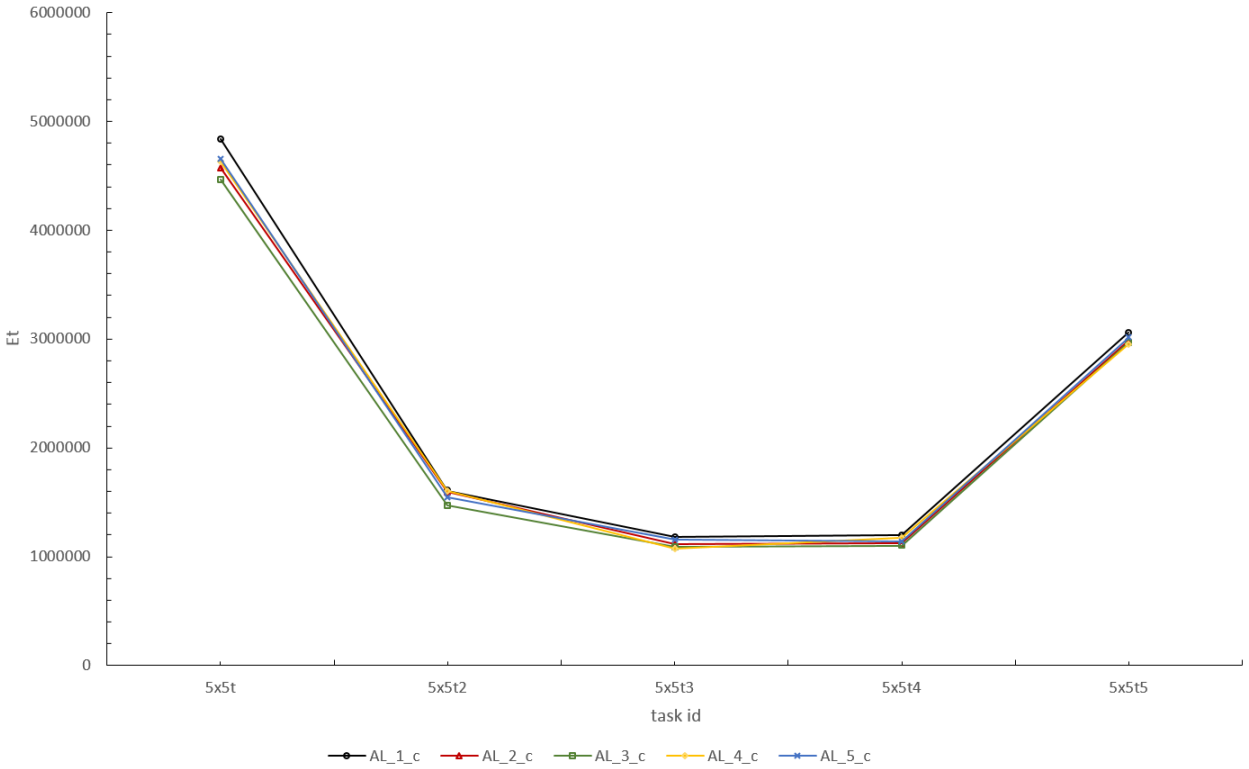


Fig. 57. Communication energy for Q = 0.5

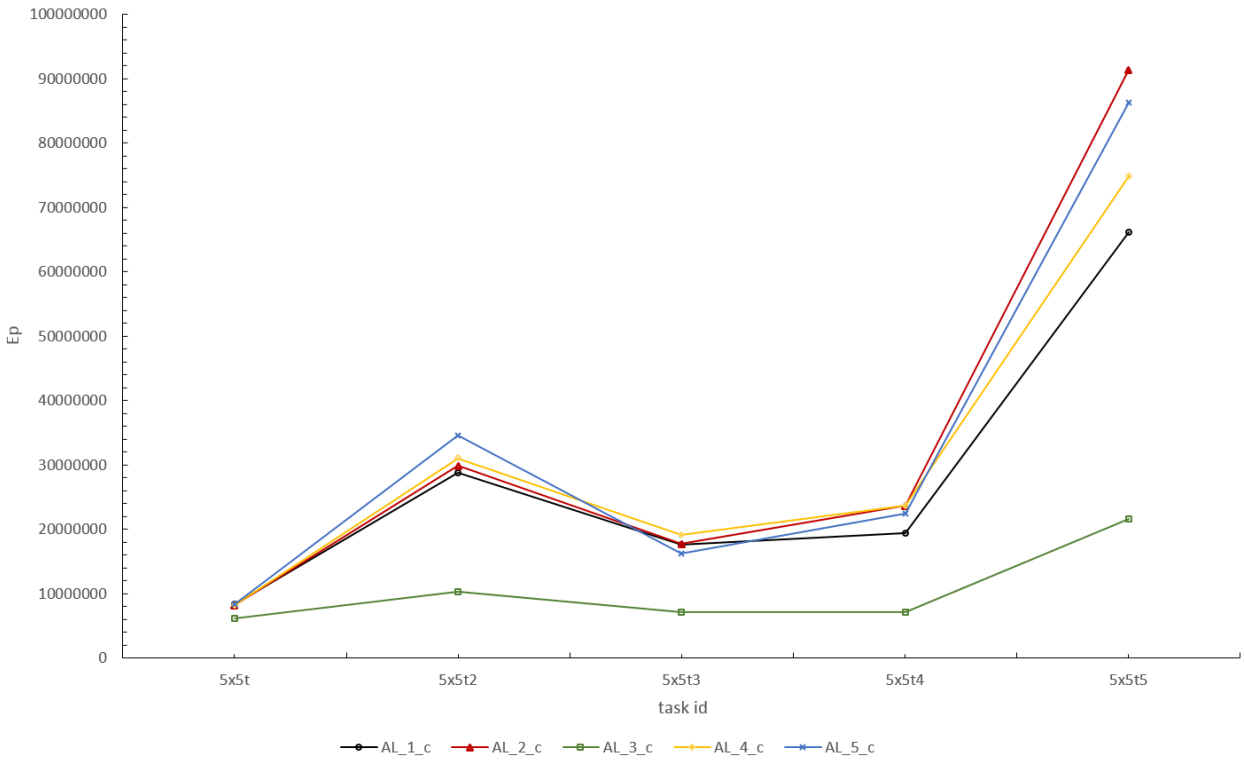


Fig. 58. Processing energy for Q = 0.5

The split of the E into communication energy and processing energy is shown in Fig. 57 and Fig. 58. As the AL_MATCH_BLOCK_TO_PU considers only the properties of the nodes and processing units, regardless of transmission parameters, the communication cost is similar for all algorithms – as shown in Fig. 57. The impact of the AL_MATCH_BLOCK_TO_PU is in the processing costs. Fig. 58 shows the efficiency of the AL_MATCH_BLOCK_TO_PU_3 algorithm, while remaining ones demonstrate similar efficiency application-to-application wise. Taking AL_MATCH_BLOCK_TO_PU_3 as a base, and using (8) for comparison, the following results appear: AL_MATCH_BLOCK_TO_PU_1 on average consumed 50% more of energy (E) than AL_MATCH_BLOCK_TO_PU_3 (minimum being 50% and maximum 75%), similar results are obtained while comparing AL_MATCH_BLOCK_TO_PU_3 to the remaining algorithms: AL_MATCH_BLOCK_TO_PU_2: average $C_{AL\_MATCH\_BLOCK\_TO\_PU\_2}^{AL\_MATCH\_BLOCK\_TO\_PU\_3}$ = 51%, min = 17%,

max = 84%; AL_MATCH_BLOCK_TO_PU_4: average $C_{AL\_MATCH\_BLOCK\_TO\_PU\_4}^{AL\_MATCH\_BLOCK\_TO\_PU\_3}$ = 51%, min = 17%, max = 77%; AL_MATCH_BLOCK_TO_PU_5: average $C_{AL\_MATCH\_BLOCK\_TO\_PU\_5}^{AL\_MATCH\_BLOCK\_TO\_PU\_3}$ = 51%, min = 17%, max = 77%. Results are summarized in Tab. 7. Regarding the time of the processing, AL_MATCH_BLOCK_TO_PU_3 is always the fastest. On average, compared to AL_MATCH_BLOCK_TO_PU_3: AL_MATCH_BLOCK_TO_PU_1 is 164% slower, AL_MATCH_BLOCK_TO_PU_2 is 182% slower, AL_MATCH_BLOCK_TO_PU_4 is 177% slower and AL_MATCH_BLOCK_TO_PU_5 is 175% slower. In some cases, time differences are much significant, i.e. AL_MATCH_BLOCK_TO_PU_2 being 820% slower than AL_MATCH_BLOCK_TO_PU_3. Tab. 6 shows the measured values. Fig. 59 shows the execution time for a system with Q = 0.8 and all the applications and algorithms.

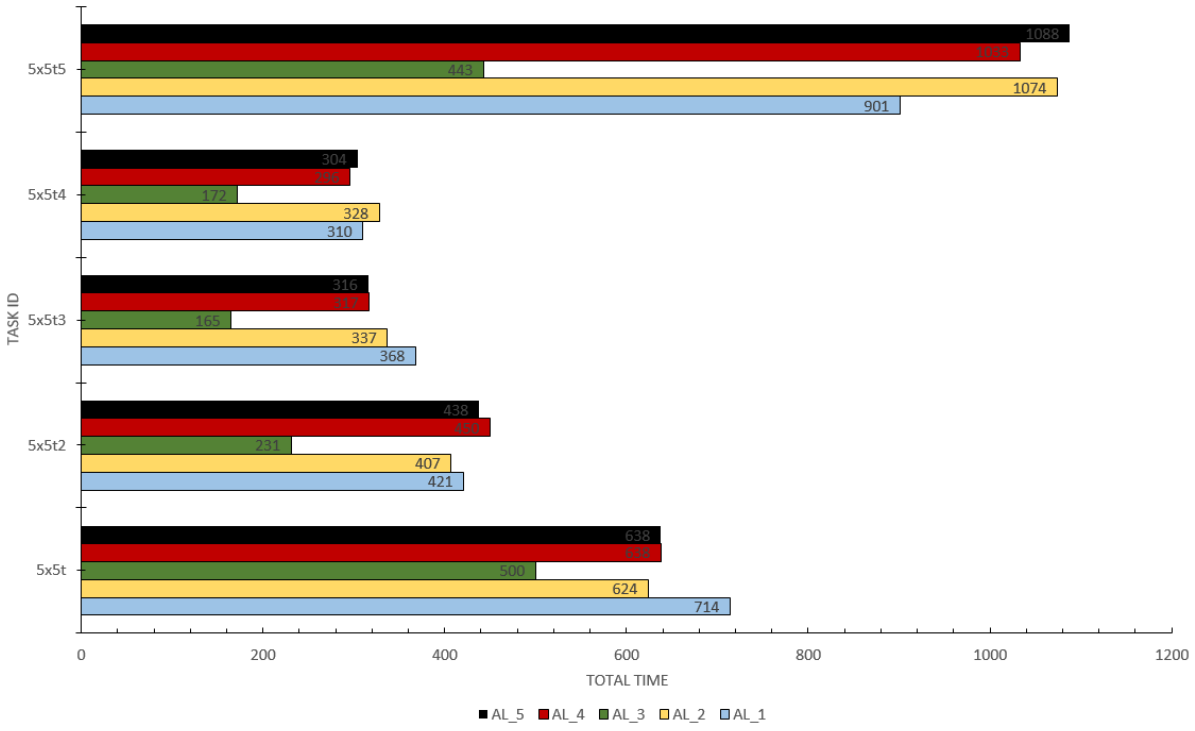


Fig. 59. Execution time for system with Q = 0.8

Table. 6. Comparison of AL_MATCH_BLOCK_TO_PU_3 to remaining algorithms (E_t)

	AL_MATCH_BLOCK_TO_PU_1 $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_1}$	**AL_MATCH_BLOCK_TO_PU_2** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_2}$	**AL_MATCH_BLOCK_TO_PU_4** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_4}$	**AL_MATCH_BLOCK_TO_PU_5** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_5}$
Time avg	164%	182%	177%	175%
Time min	35%	20%	22%	23%
Time max	485%	820%	548%	544%

Table. 7. Comparison of AL_MATCH_BLOCK_TO_PU_3 to remaining algorithms (E_p)

	AL_MATCH_BLOCK_TO_PU_1 $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_1}$	**AL_MATCH_BLOCK_TO_PU_2** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_2}$	**AL_MATCH_BLOCK_TO_PU_4** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_4}$	**AL_MATCH_BLOCK_TO_PU_5** $C^{AL\_MATCH\_BLOCK\_TO\_PU\_3}_{AL\_MATCH\_BLOCK\_TO\_PU\_5}$
Cost avg	50%	51%	51%	51%
Cost min	20%	17%	17%	17%
Cost max	75%	84%	77%	77%

The Q value defined in (17) is observed to impact the processing cost E, however Q is not something that the system operation can have influence on – as Q depends mainly on the hardware parameters of each node. Therefore, it is essential that operating algorithms and overall system design operates efficiently regardless of Q (regardless of the hardware structure given). Hence, the experiments have been performed to research the impact of Q on AL_MATCH_BLOCK_TO_NODE algorithms, as matching block to nodes shows the highest sensitivity to change in hardware parameters. Experiments have been done for 5 applications, characterized in Tab. 5 – to cover various types of applications.

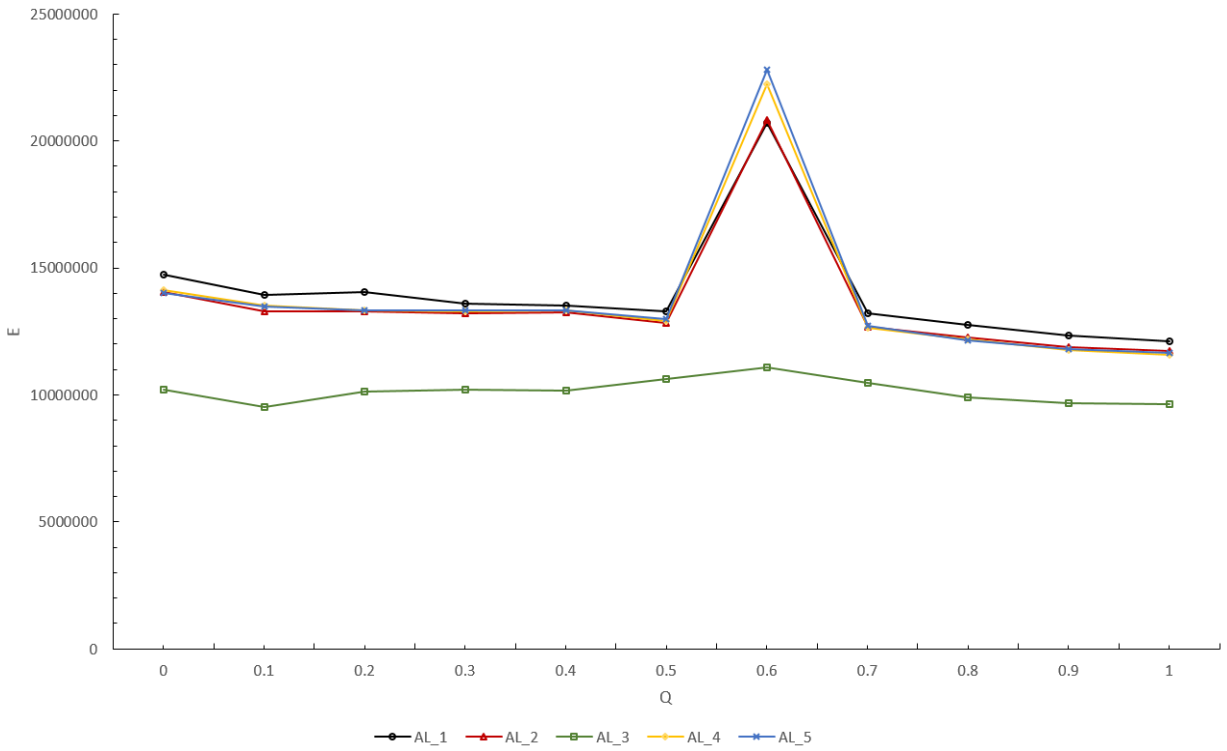


Fig. 60. Relation of *Q* to *E*, application *5x5t1*

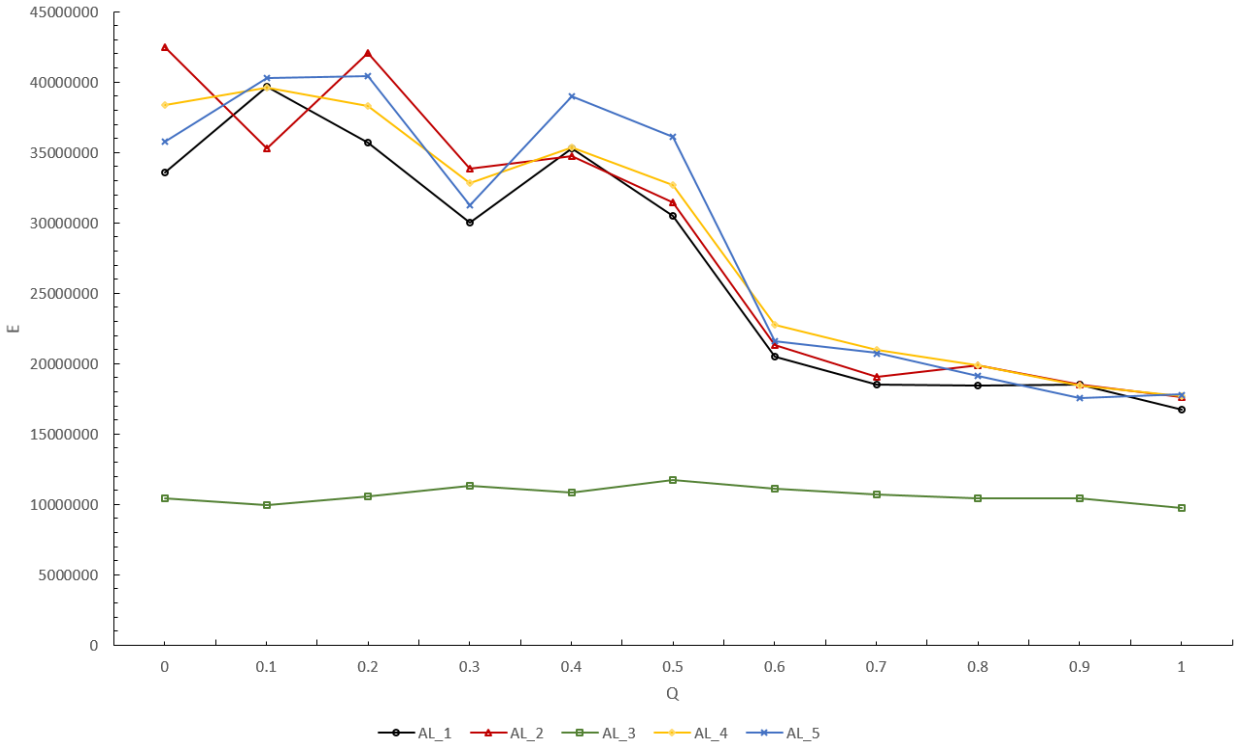


Fig. 61. Relation of *Q* to *E*, application *5x5t2*

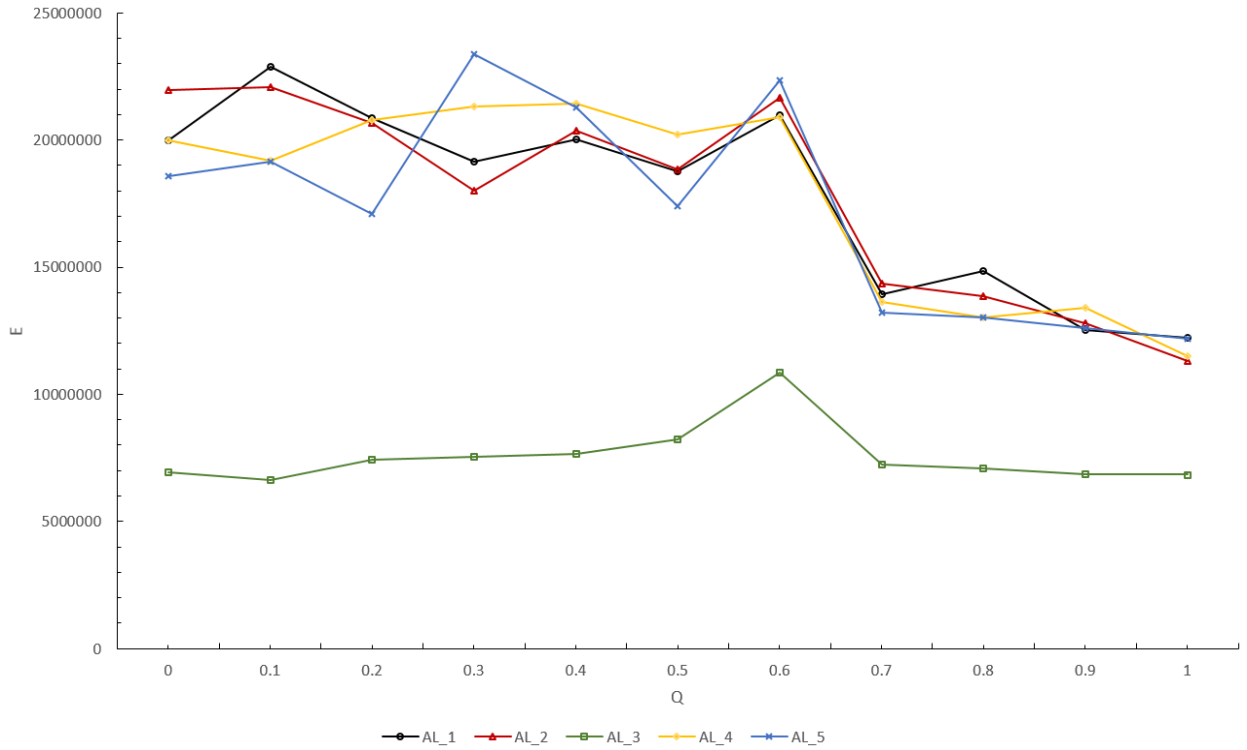


Fig. 62. Relation of *Q* to *E*, application *5x5t3*

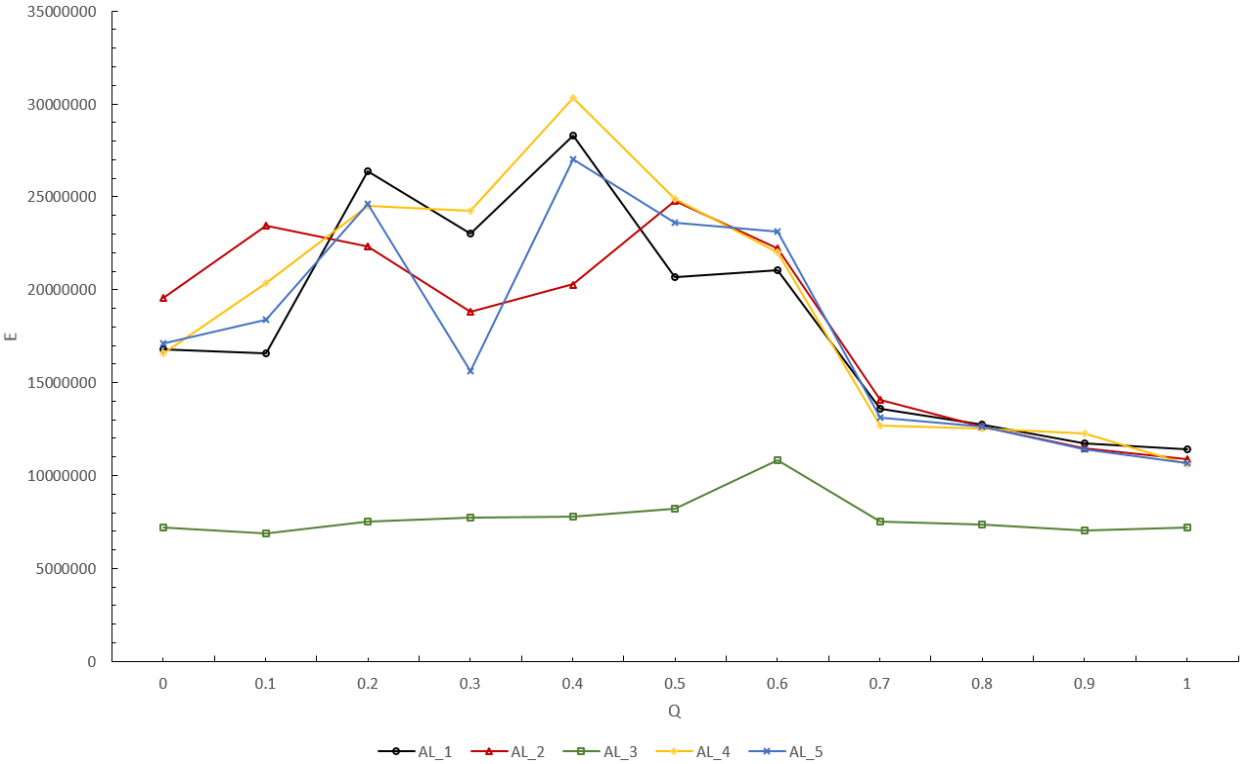


Fig. 63. Relation of *Q* to *E*, application *5x5t4*

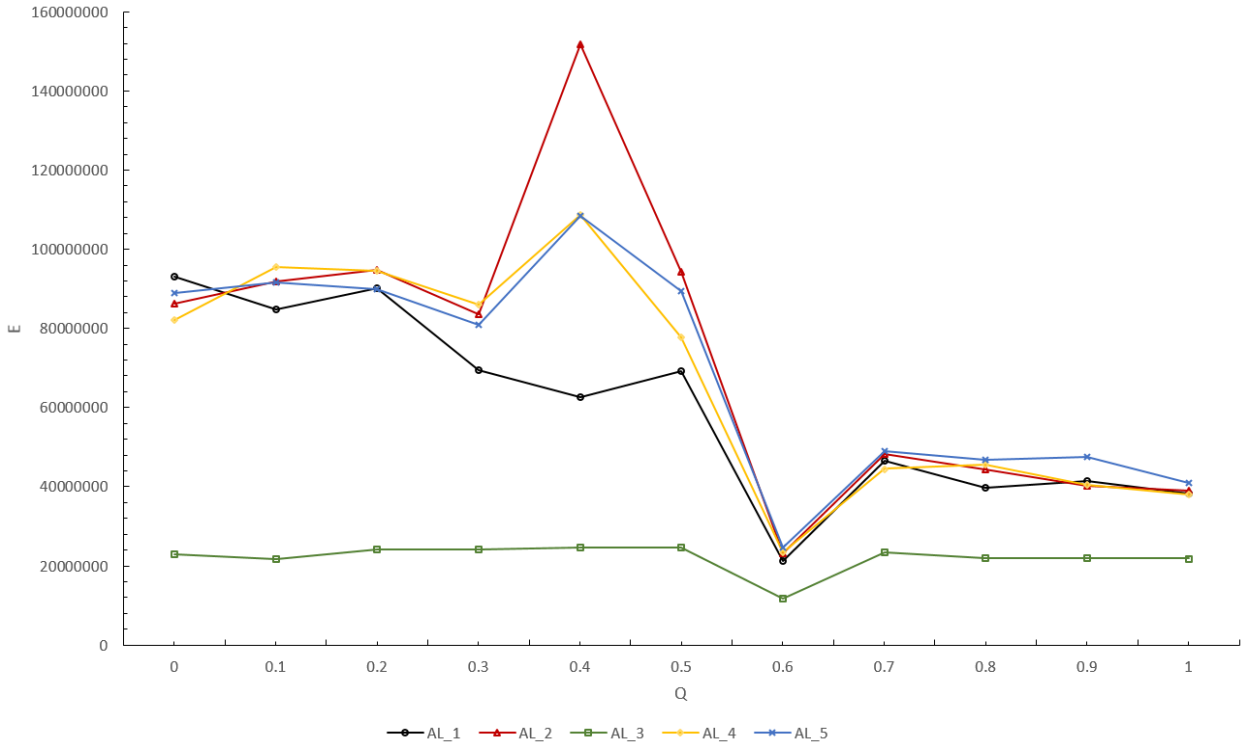


Fig. 64. Relation of *Q* to *E*, application *5x5t5*

Experimentation show that AL_MATCH_BLOCK_TO_PU_3 always yields the smallest *E*, for all types of applications. The value of 0.6 appeared to be a point of special impact to *E* – the results for AL_MATCH_BLOCK_TO_PU_3 follow results for the remaining algorithms, but keep the lower *E* value in any case. Results are presented in figures Fig. 60 – Fig. 64.

Time of processing is also affected by *Q* value and research experiments showed that for four types of applications (5x5t2, 5x5t3, 5x5t4, 5x5t5 – Fig. 66 – Fig. 69) the processing time very often proportionally resembles the processing cost for the same application and the same algorithm. Again, the AL_MATCH_BLOCK_TO_PU_3 is always the fastest. For application type of 5x5t1 (small number of online blocks, moderate size of online blocks, large number of defined blocks, small size of defined blocks, application size 143MB) the relation between processing time and *E* are different (Fig. 65) – and also contradicting for the case of *Q* = 0.6.

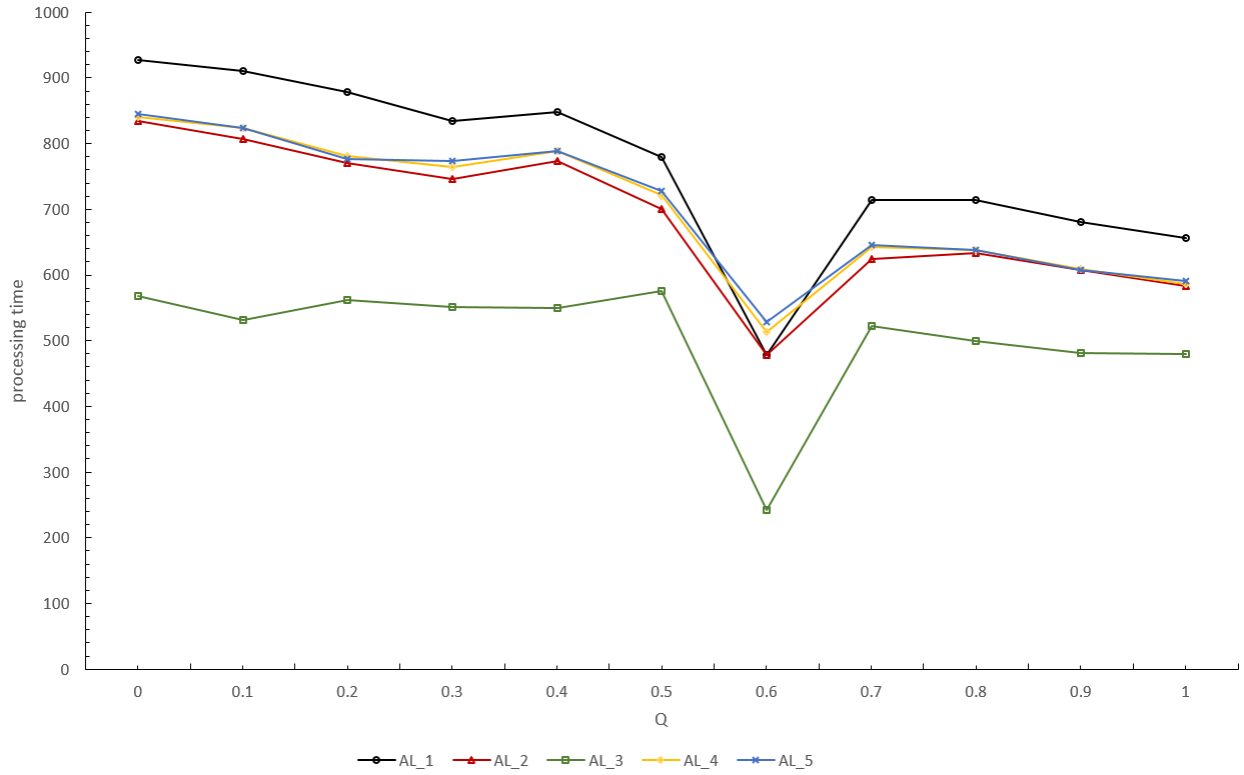


Fig. 65. Relation of *Q* to processing time, application *5x5t*

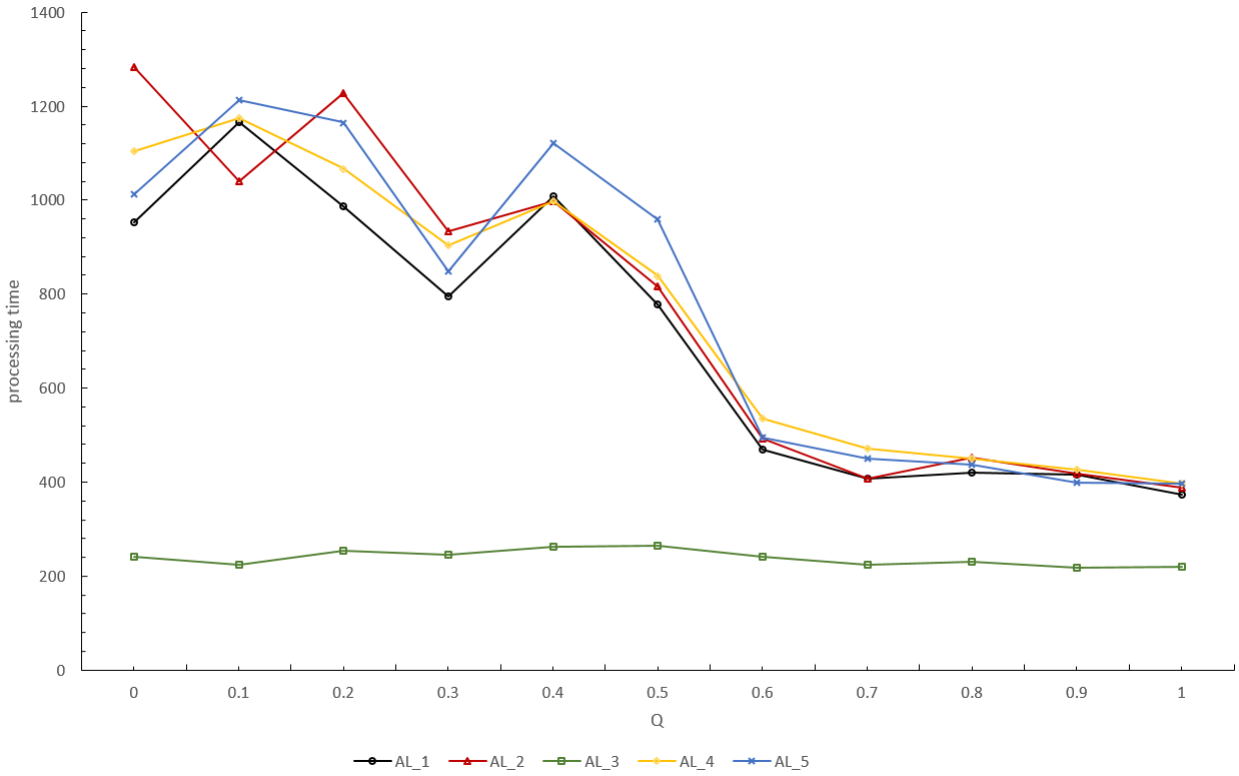


Fig. 66. Relation of *Q* to processing time, application *5x5t2*

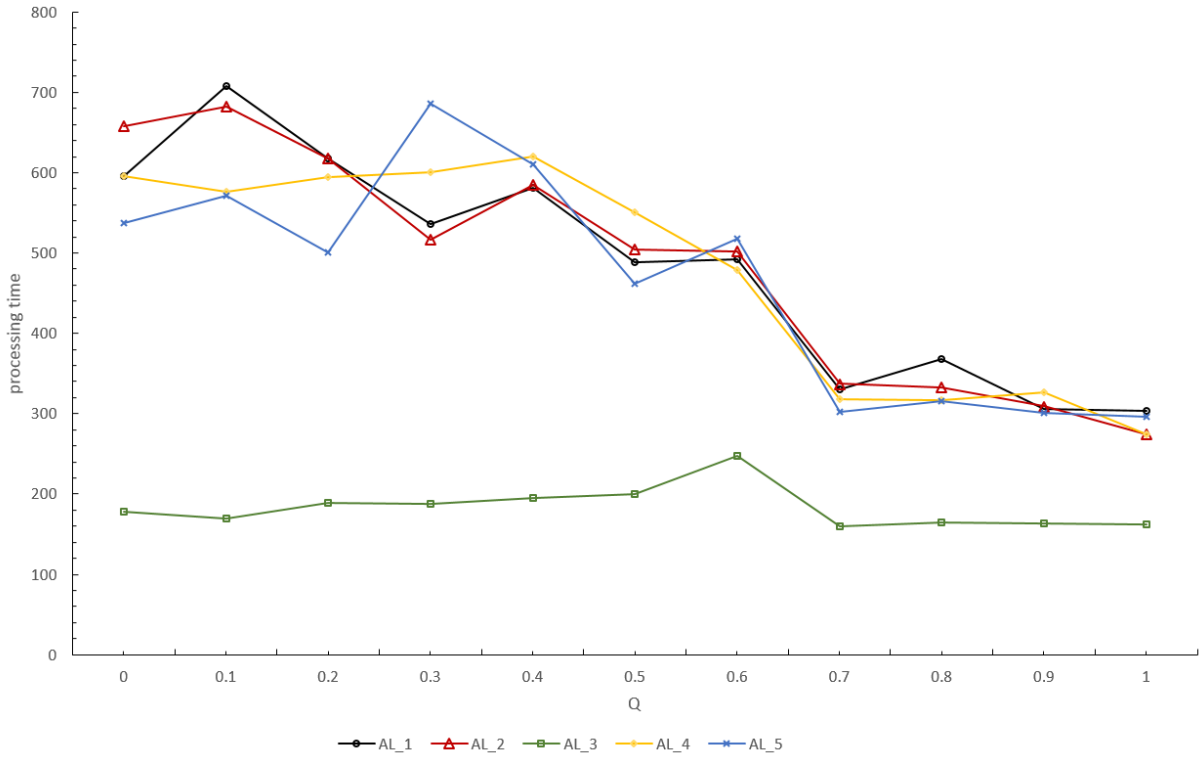


Fig. 67. Relation of *Q* to processing time, application *5x5t3*

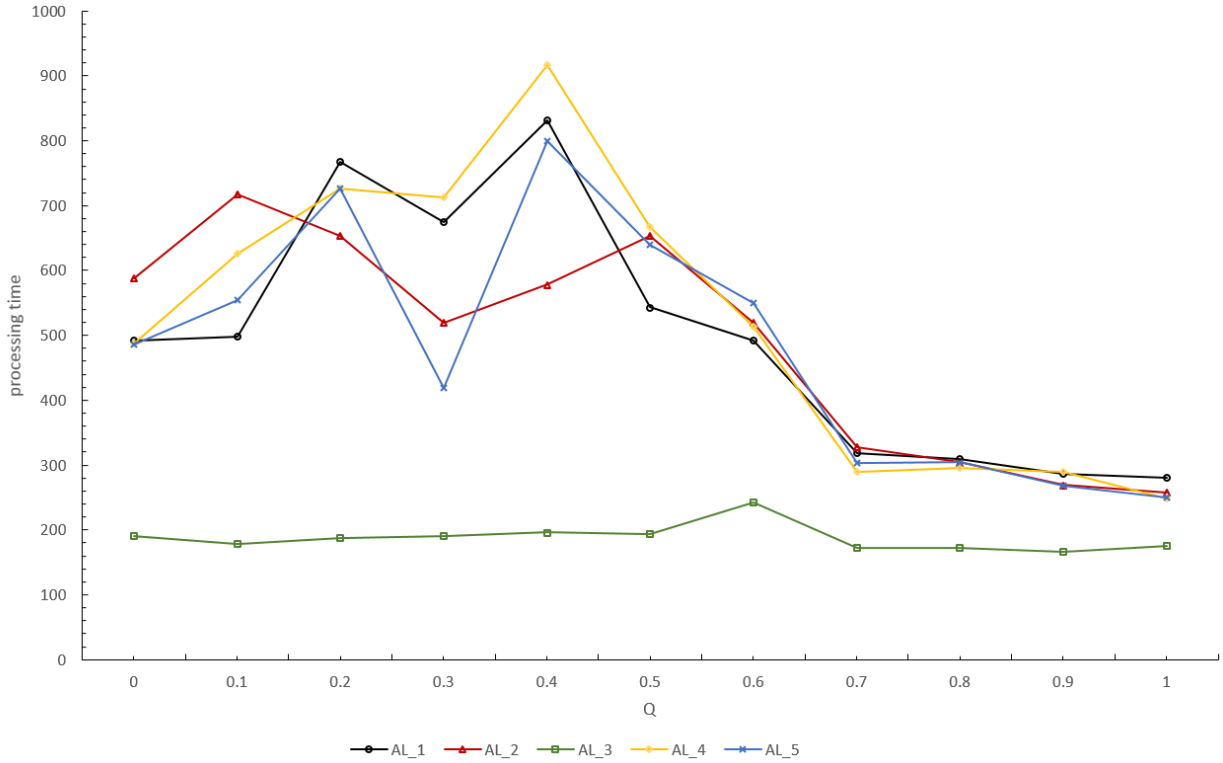


Fig. 68. Relation of *Q* to processing time, application *5x5t4*

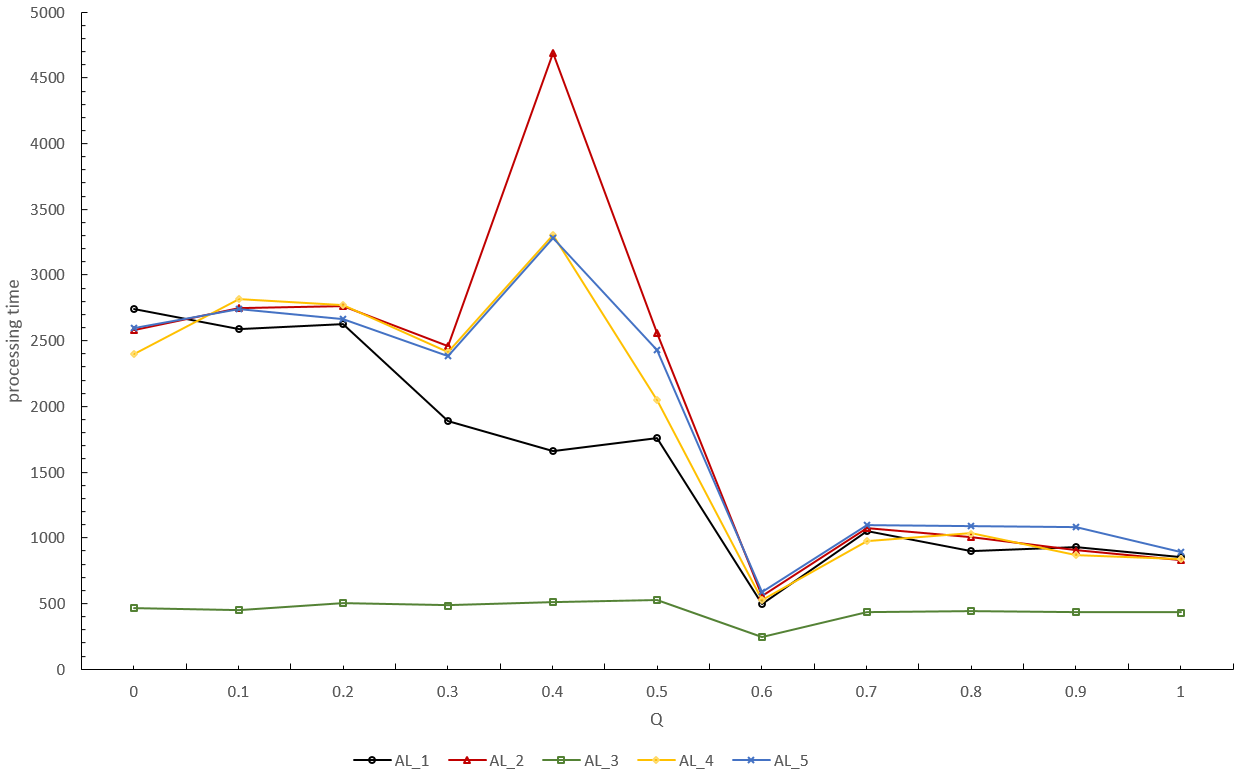


Fig. 69. Relation of *Q* to processing time, application *5x5t5*

Experiments performed on other applications and systems confirmed the results described above.

The Gantt graph presented in Fig. 74 (Appendix) shows the detailed system operation for the same system and application, and two algorithms: AL_MATCH_BLOCK_TO_PU_1 and AL_MATCH_BLOCK_TO_PU_3. Each type of operation is represented: adding control nodes, obtaining knowledge about tasks, getting block (matching + downloading from task owner), getting datasources, getting functions, processing block, uploading result, processing block online, deciding reconfiguration, downloading bitstream and programming bitstream. The node/processing unit utilization over total simulation time can be observed: the processing time for AL_MATCH_BLOCK_TO_PU_1 is not even twice as long as AL_MATCH_BLOCK_TO_PU_3 (974 slots vs 575 slots), but also processing units are better utilized (Tab. 8). The average processing unit utilization is 43.5% for

AL_MATCH_BLOCK_TO_PU_1 and 63.6% for AL_MATCH_BLOCK_TO_PU_3. The minimum, maximum and median values are 11.7%, 96.6%, 43.1% and 63.6%, 14.7%, 99.4%, 54%, for former and latter cases.

Table. 8. The utilization of the processing units

	AL_MATCH_BLOCK_TO_PU_1	**AL_MATCH_BLOCK_TO_PU_3**
avg	43.5%	63.6%
min	11.7%	14.7%
max	96.6%	99.4%
median	43.1%	54%

Table Tab. 9 presents the average percentage use over all processing units, for each operation. Most of the time spent is on getting the block (that includes matching + downloading from task owner), and depends on the communication speeds. The most important factor is the time spent for processing – which on average reached 63.6% for AL_MATCH_BLOCK_TO_PU_3 and 43.5% for AL_MATCH_BLOCK_TO_PU_1. This results show the efficiency of the AL_MATCH_BLOCK_TO_PU_3 algorithm. All of the values for AL_MATCH_BLOCK_TO_PU_3 are better than for AL_MATCH_BLOCK_TO_PU_1. Getting block, getting datasources, getting functions, processing block, uploading result, processing block online are all higher for AL_MATCH_BLOCK_TO_PU_3. This means the better utilization of the processing units and therefore higher efficiency. Lower values of deciding reconfiguration, downloading bitstream and programming bitstream mean that the processing unit / node spends less time for non-processing self-management tasks, therefore improving the efficiency. Adding control nodes and obtaining knowledge about tasks are performed at the level of the entire node and are not considered to be related specifically to processing units. The detailed values of the average utilization per operation is shown in Tab. 9, Fig. 70 and Fig. 71. Average number of slots

for each operation is calculated for each processing unit/node separately, and then these values are averaged over all pu:s/nodes, resulting in average time slots that the processing unit/node spends on each of the operations.

Table. 9. The utilization of the processing units (averaged values)

Operation	AL_MATCH_BLOCK_TO_PU_1	AL_MATCH_BLOCK_TO_PU_3
adding control nodes	0.3%	0.51%
obtaining knowledge about tasks	0.21%	0.36%
getting block	13.94%	20.42%
getting datasources	4.2%	7%
getting functions	6.69%	11.61%
processing block	7.7%	10.36%
uploading result	8.54%	13.02%
processing block online	0.92%	0.94%
deciding reconfiguration	0.13%	0.08%
downloading bitstream	0.14%	0.09%
programming bitstream	0.25%	0.15%
total	**43.06%**	**64.58%**

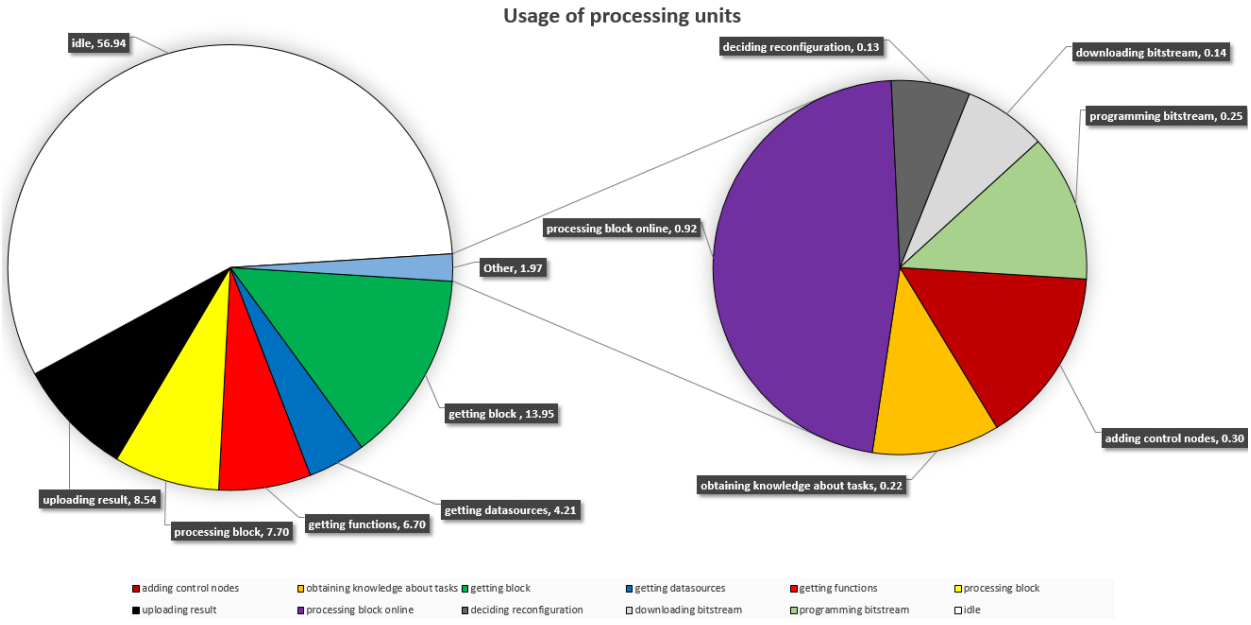


Fig. 70. Usage of processing units, AL_MATCH_BLOCK_TO_PU_1

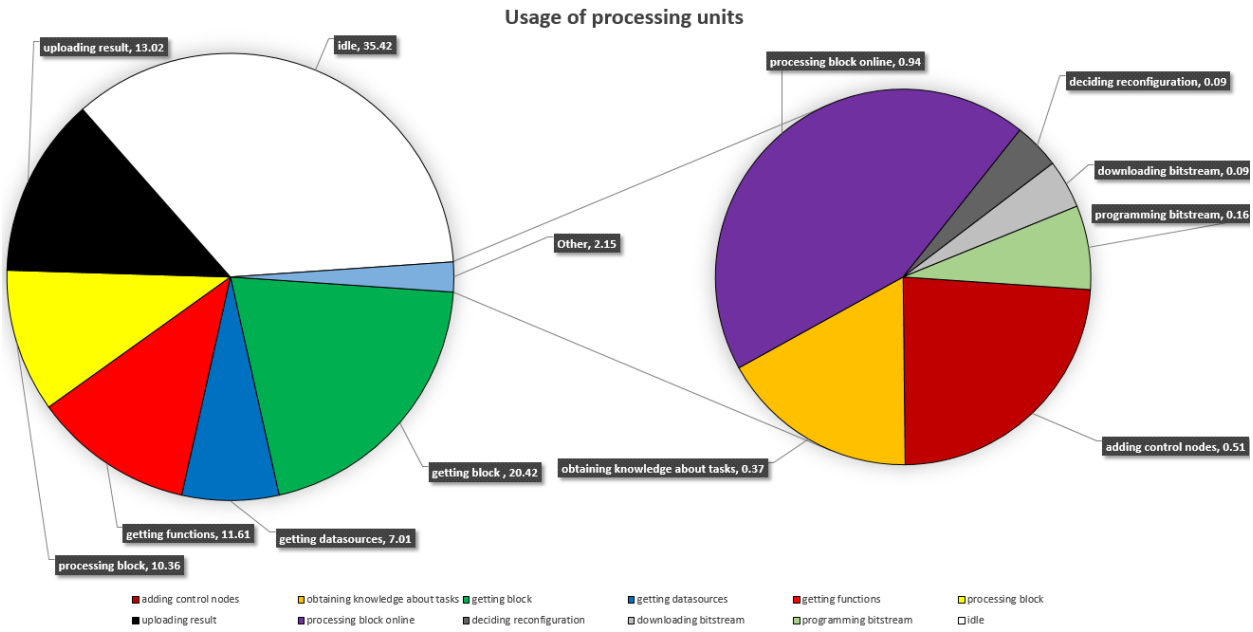


Fig. 71. Usage of processing units, AL_MATCH_BLOCK_TO_PU_3

5.4.6. Summary of efficiency gain

The most important outcome of system design and optimization is the overall efficiency gain compared to non-optimized case. In this section, the whole optimization is used (i.e. the set of most efficient algorithms, as proved in earlier sections) compared to a non-optimized case. The following system architecture was used: $V = 50$, processing units per node: 1-3, datasources per node: 1-4. Application: 46MB size, 50 online blocks (1.7MB), 500 blocks defined (43MB), size of raw data: 1MB. Cooperative blocks impact a system mostly the same way as online blocks, therefore are present as online blocks. The above system and application is selected for simulation as the one that fully and significantly reflects all the application properties, so the results for each property is clearly observed. Also, the results discussed below comply with results from the other analyzed systems and applications, so they describe the typical DPRS operation.

Figure Fig. 75 (Appendix) presents the Gantt graph comparing non-optimized (a), upper) and optimized (b), lower) cases. In this scenario, the *online blocks* are playing significantly visible role (depicted with the violet color). Comparing non-optimized (Fig. 75 a)) and optimized operations (Fig. 75 b)), much shorter operation timespan is required for b). Also much bigger utilization rate is observed for b). Same reconfiguration period algorithm is used and the RECONFIGURATION_PERIOD is set to 17 using (11). For b), the total operation time is T_b = 195, and for a) T_a = 350 (subscripts indicate a) and b) cases), meaning $C_a^b = 44.28\%$ and optimized case being over 44% faster. Communication cost is around the same value in both cases: $E_{t,a}$ = 1080262 vs. $E_{t,b}$ = 1006180 – this cost is mostly determined by the configuration of ROLE_TASK_OWNER node, and is similar as the focus of the algorithms is on the processing energy. The biggest profit appears in the processing energy: $E_{p,a}$ = 12698746 vs. $E_{p,b}$ = 8037641 – meaning $C_a^b = 36.7\%$ less power consumption for the optimized case.

Regarding node operations, Fig. 72 and Fig. 73 present the distribution of the operations for a) and b) cases respectively. Same as in Fig. 70 and Fig. 71, the operations are averaged using the same procedure. For a), the large idle time can be observed (17%) while in the optimized case b) idle stays below 1%. The share for processing blocks online is very similar (differences are the result of different moments of time when online blocks started processing, while they process till the end of the task processing). The percent of the processing blocks is also higher for b). Many percentages are higher for b), as the idle is very low and compared to a), idle slots are used for other operations.

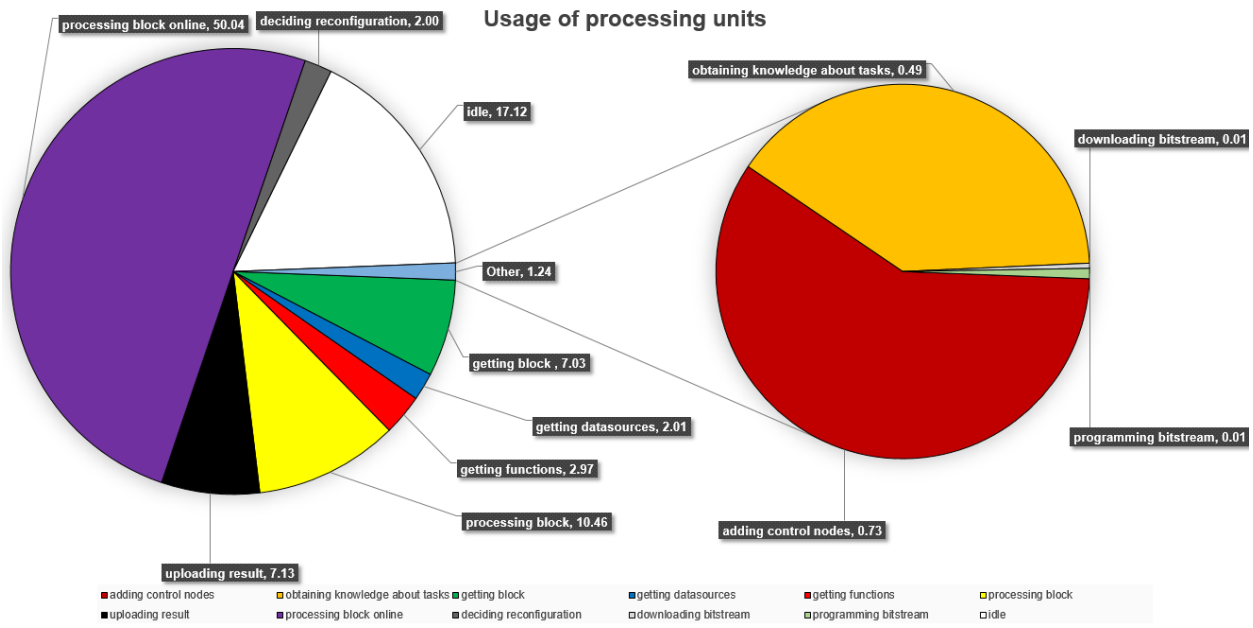


Fig. 72. Distribution of operations for the non-optimized case

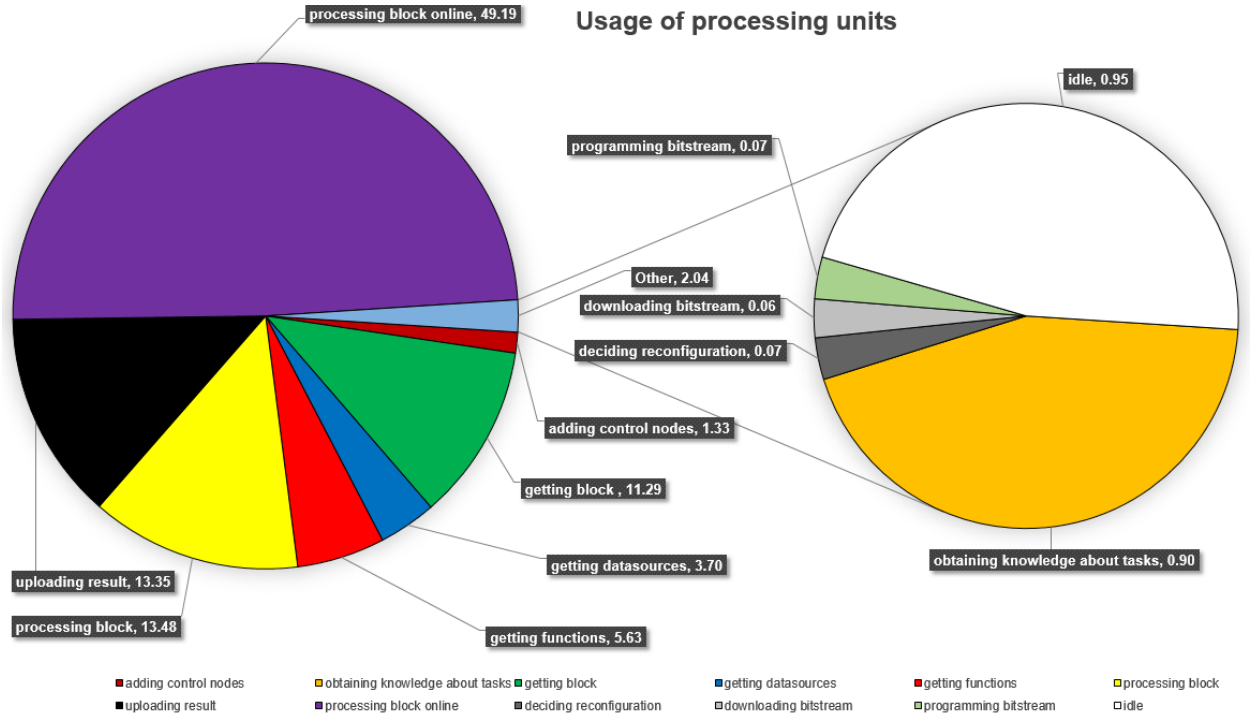


Fig. 73. Distribution of operations for the optimized case

Regarding utilization averages over processing units, for a) the minimum utilization rate is 50% and 36% for b), and maximum utilization rate 99% for both cases (processing units that process online blocks have this kind of utilization rate).

The energy expenditure elements (Tab. 2) are shown in Tab. 10. The most significant expenditure is the processing of block and the constant node operation cost. In the optimized case, nodes manage the reconfiguration more efficiently – 47% more energy is spent in the non-optimized case. Also the cost of getting blocks is lower for b).

Table 10. The distribution of the expenditure elements.

Expenditure element	**non-optimized a)**	**optimized b)**
SIM_OPEX_C_ADDING_CONTROL	3450	3450
SIM_OPEX_T_GETTING_TASKS	2397	2397
SIM_OPEX_G_GETTING_BLOCK	478186	402210
SIM_OPEX_D_GETTING_DS	19362	18796
SIM_OPEX_F_GETTING_FN	29423	29946
SIM_OPEX_U_UPLOADING_RESULT	547367	549117
SIM_OPEX_R_FPGA_DECIDE_RECONF	228600	120450
SIM_OPEX_1_NODE_PER_SLOT	1654576	902496
SIM_OPEX_P_PROCESSING_BLOCK	10815550	6008450

CHAPTER 6

CONCLUSIONS

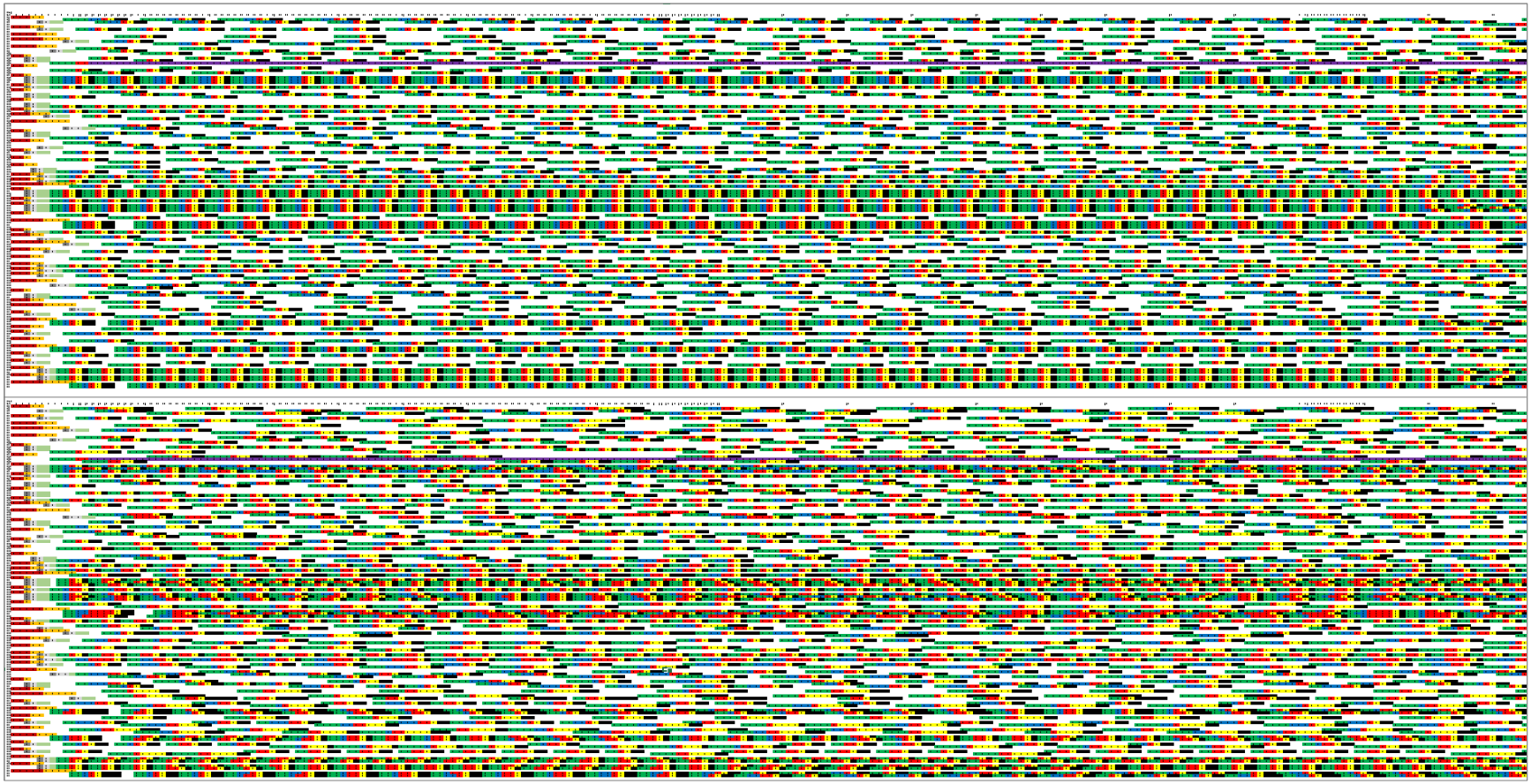
The problem of optimization in distributed processing systems with reconfigurable computing elements has been presented and addressed in this work. Hardware architecture has been proposed in detail, providing a comprehensive description of the solution. Logical formats of the elements have also been designed and presented. The physical and logical structures have been designed in a layered and modular fashion, so that the operational algorithms possessed the flexibility to replace and use any transmission layer and/or data format. In addition, the operational algorithms have been designed, described and experimented. The proposed solutions have been tested both separately and together (when possible), to clearly show their impact. Next key aspect of the proposed work is the design of the application definition that allows to describe various types of applications and execute them in the distributed DPRS environment. All those elements together, proposed, designed and tested in this work, constitute a complete distributed processing system with the ability to reconfigure, which is also partly decentralized and provides extensive autonomy to its components.

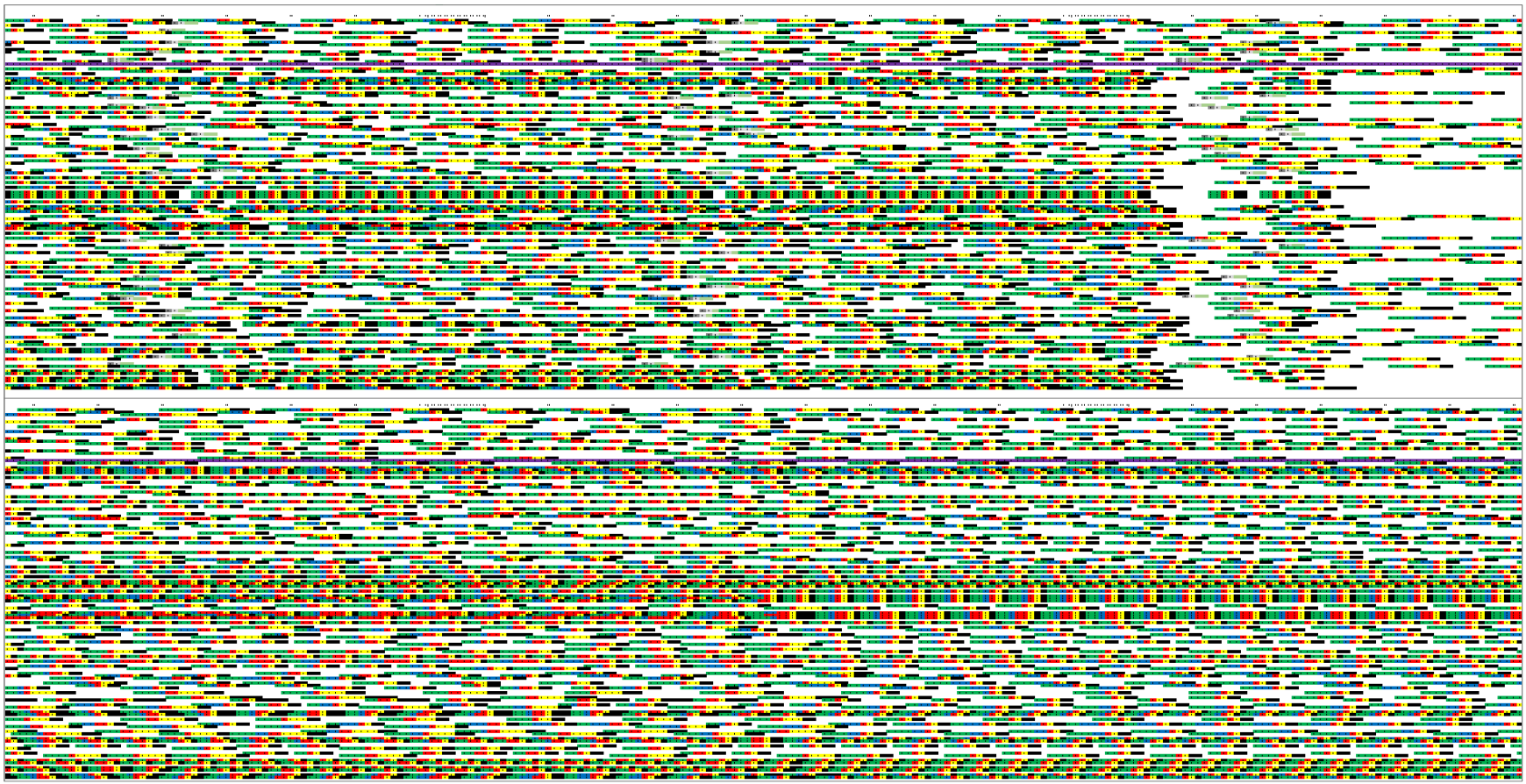
The investigated system included multiple mechanisms and most of the parts and mechanisms could be optimized using separate algorithms. Lowering the operational cost in one area could lead to cost increase in another, or cause difficulty in the other parts' optimization. Therefore, the design of the operating principles and algorithms, as well as the system structure is very challenging. Despite these elements, the system parameters and applications' properties are also greatly influencing system operation and energy expenditure. Many specific systems and many specific applications could be much better optimized using dedicated mechanisms, algorithms and system

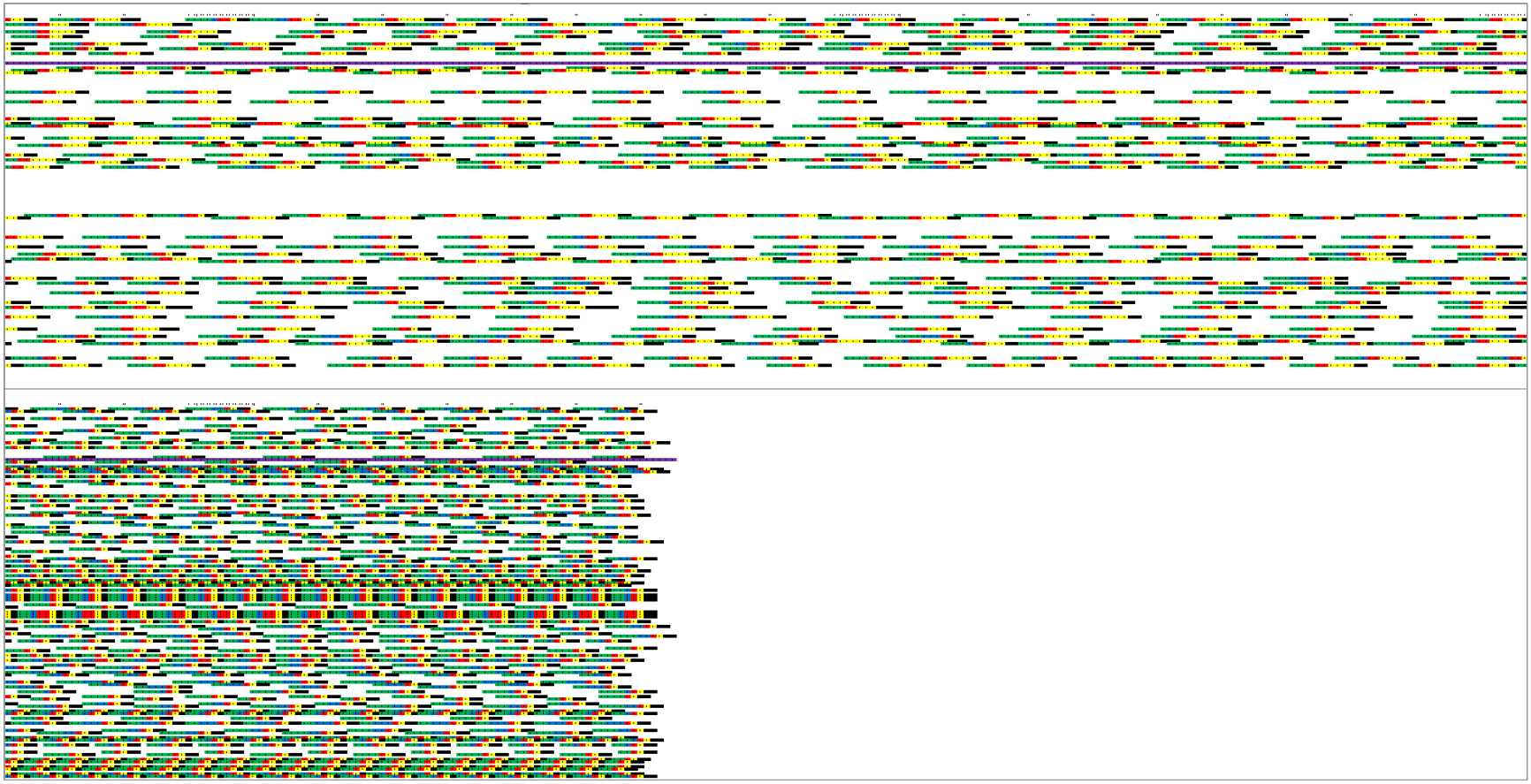
architecture. It is impossible, or often unprofitable to use this approach. The architecture and solutions presented in this work have been designed to be able to handle most cases and to be universal. The main goal has been to minimize the operational energy in the reconfiguration-capable system. The DPRS system has possible application in many modern engineering areas, such as Internet of Things, UAV cooperative teams and distributed computation systems containing various processing devices including reconfigurable FPGA chips. Multiple algorithms have been designed and tested for the purpose of this work, and their operation and efficiency have been compared with non-optimized algorithms. Results demonstrate that the proposed algorithms provide significant operational energy savings and shorten the processing time. Such profits are invaluable in the case of battery powered systems, and other systems that do not have constant power source (also UAVs using combustible engines).

Future work is to extend the DPRS architecture with task migration architecture (ability for one task owner to pass the task to another node while task is being processed), advanced synchronization methods for control nodes (including the strong reliability mechanisms), adding the security layer, and dynamic reconfiguration period algorithms. The multi-systems cooperation is an important element to consider in future work, followed by incentive mechanism that would play a significant role in such environment.

APPENDIX







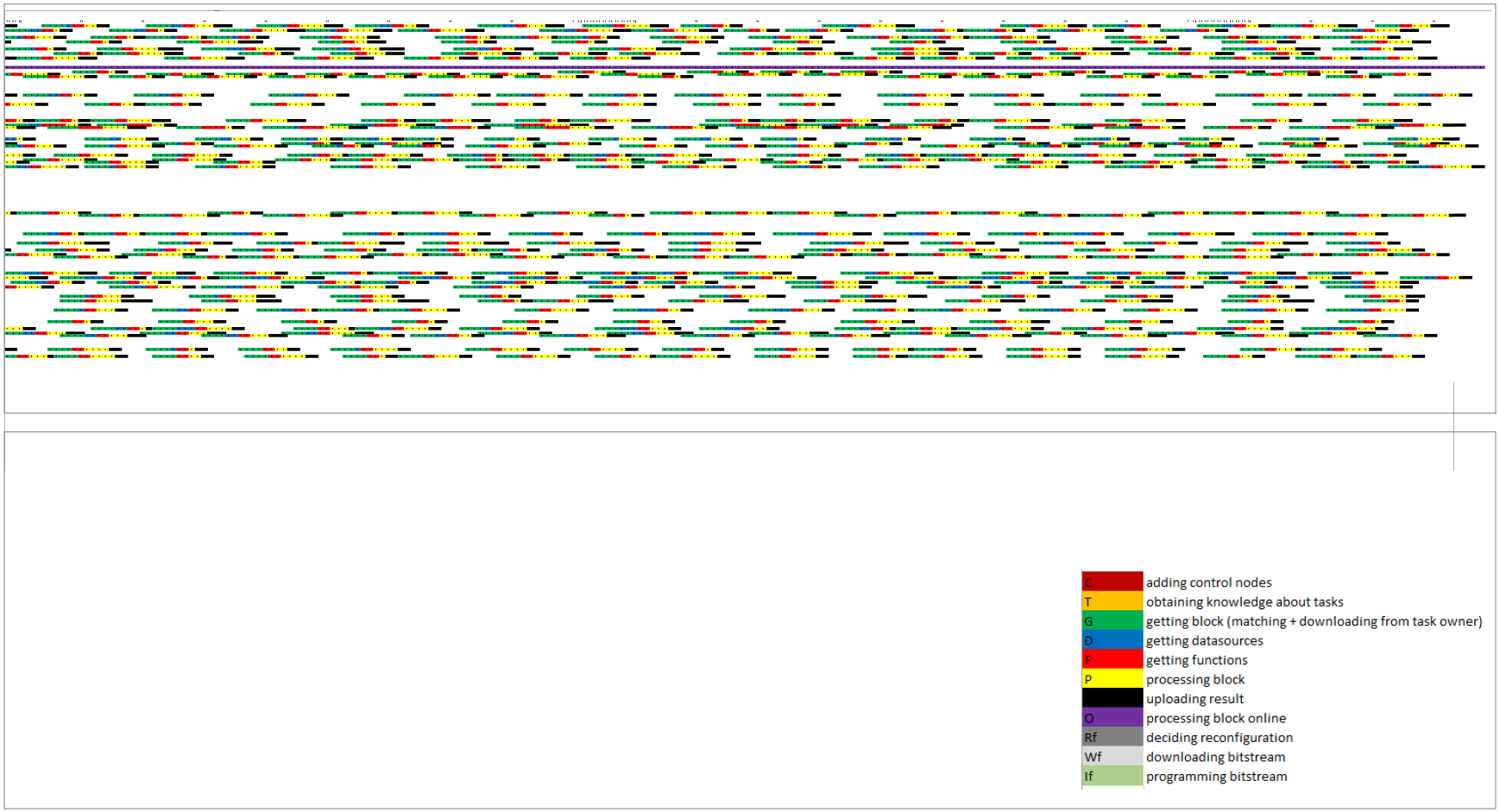
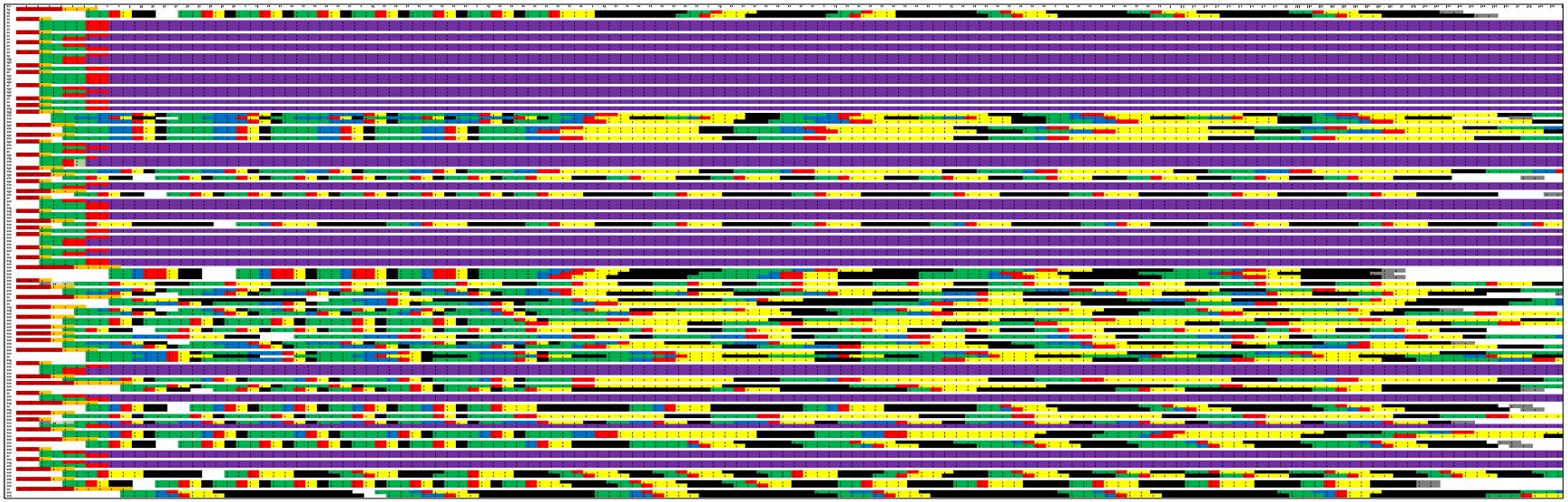
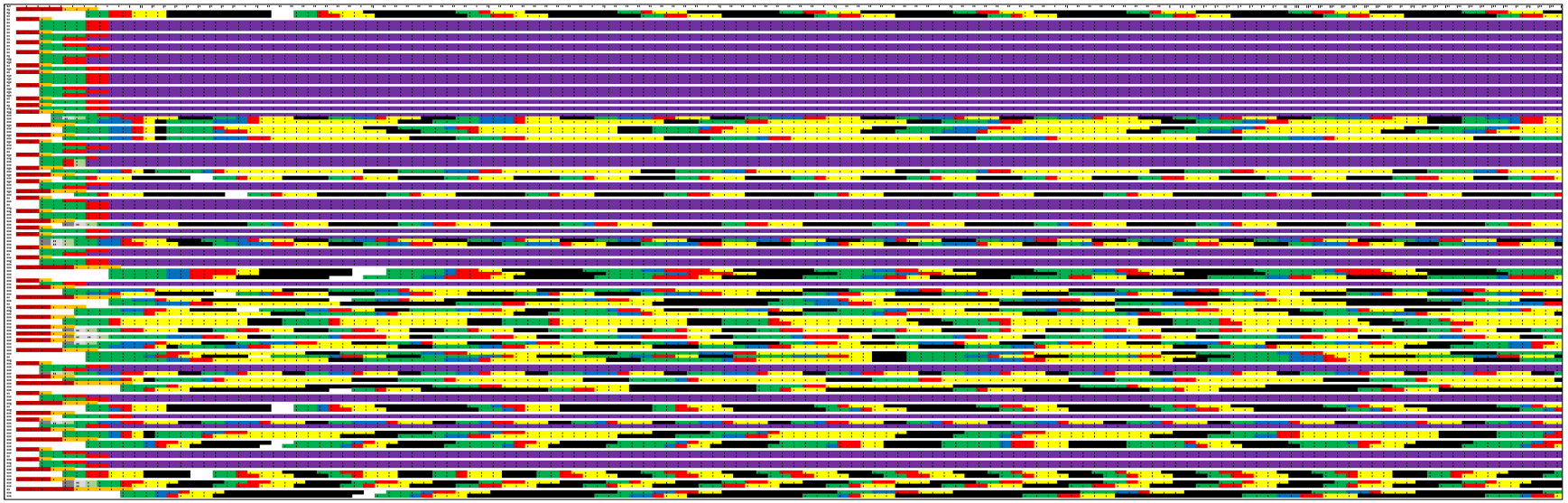


Fig. 74. Gantt graph: System operation for AL_MATCH_BLOCK_TO_NODE_1 and AL_MATCH_BLOCK_TO_NODE_3

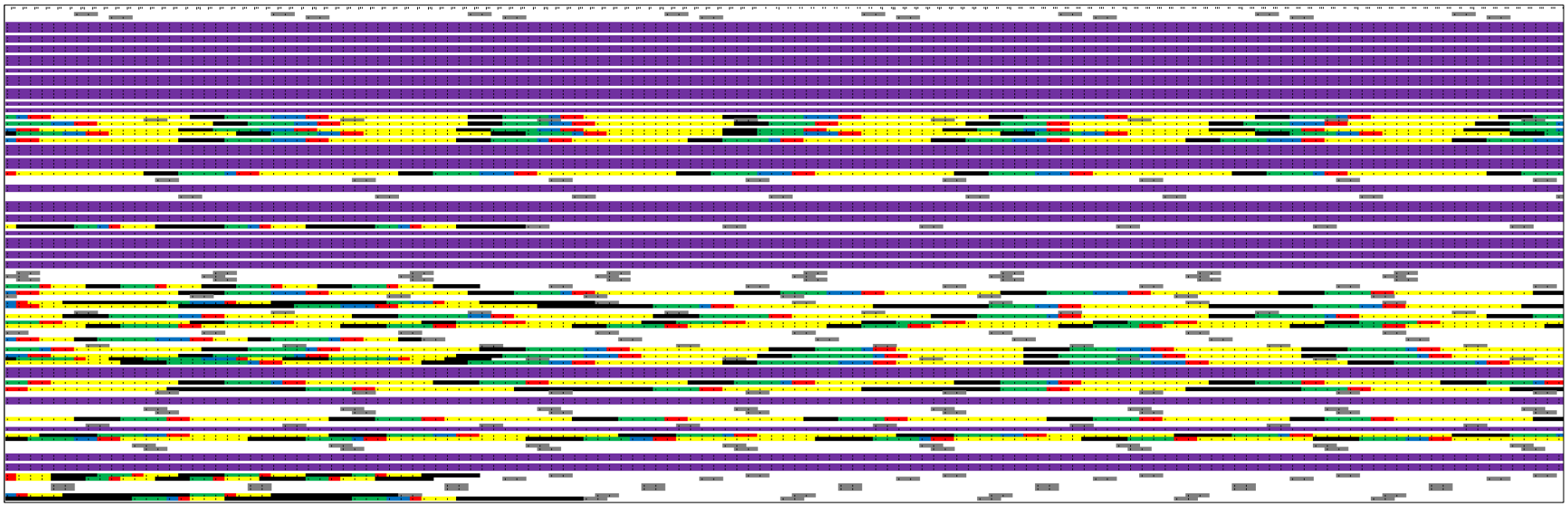
a)



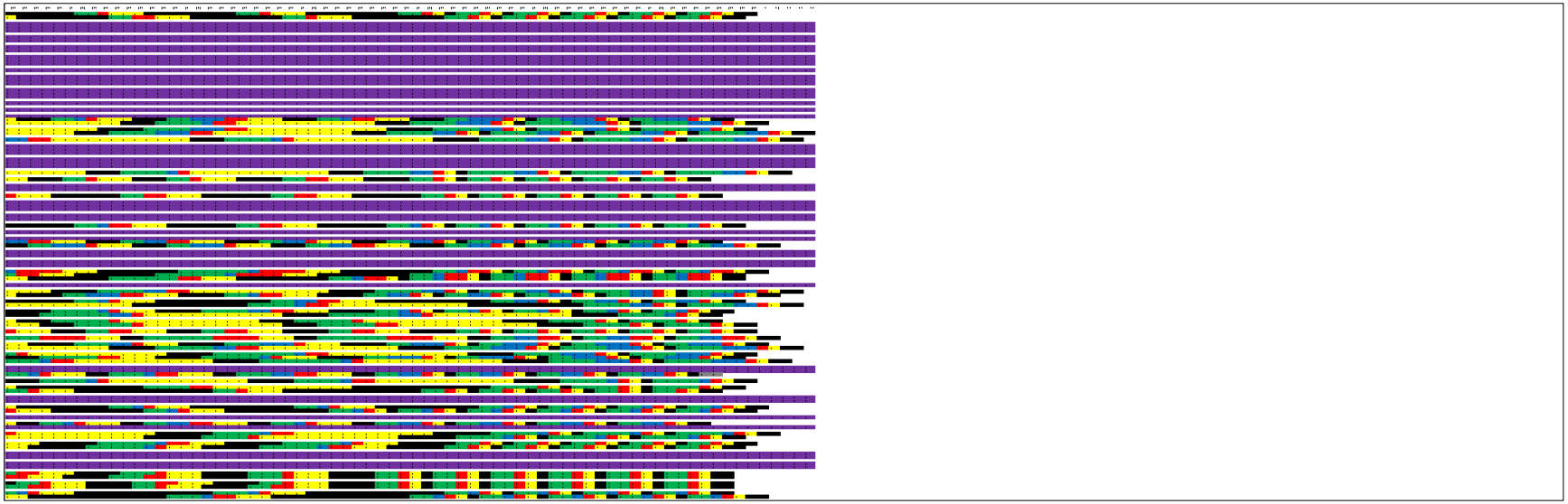
b)



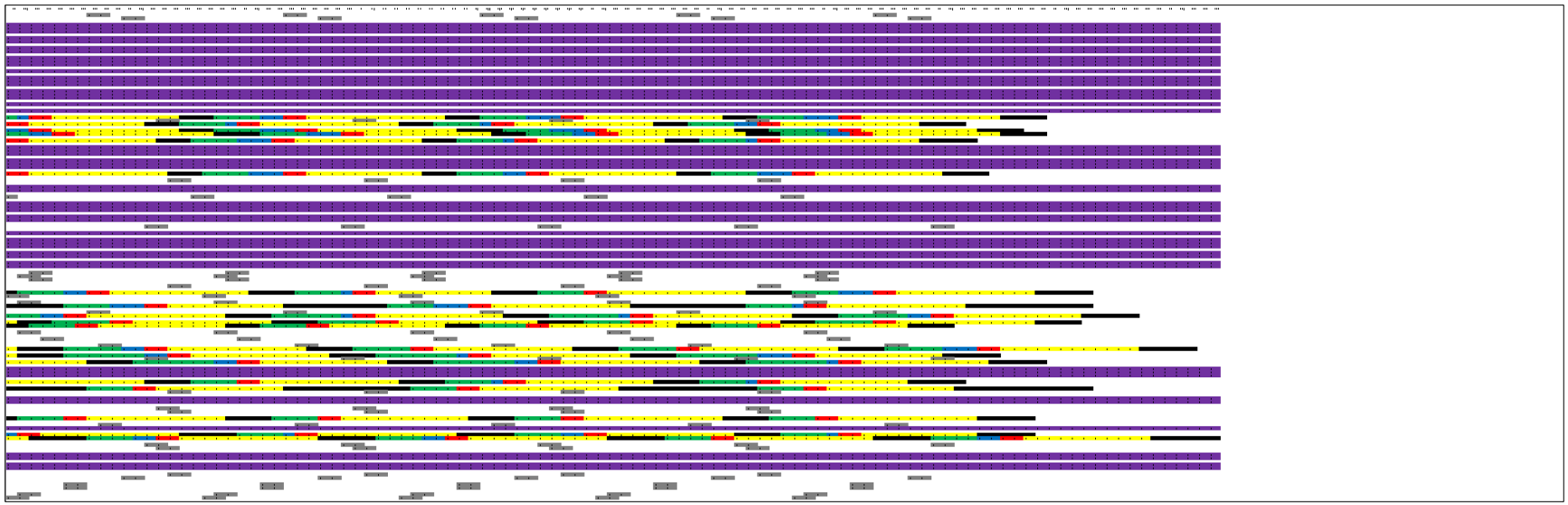
a)



b)



a)



b)

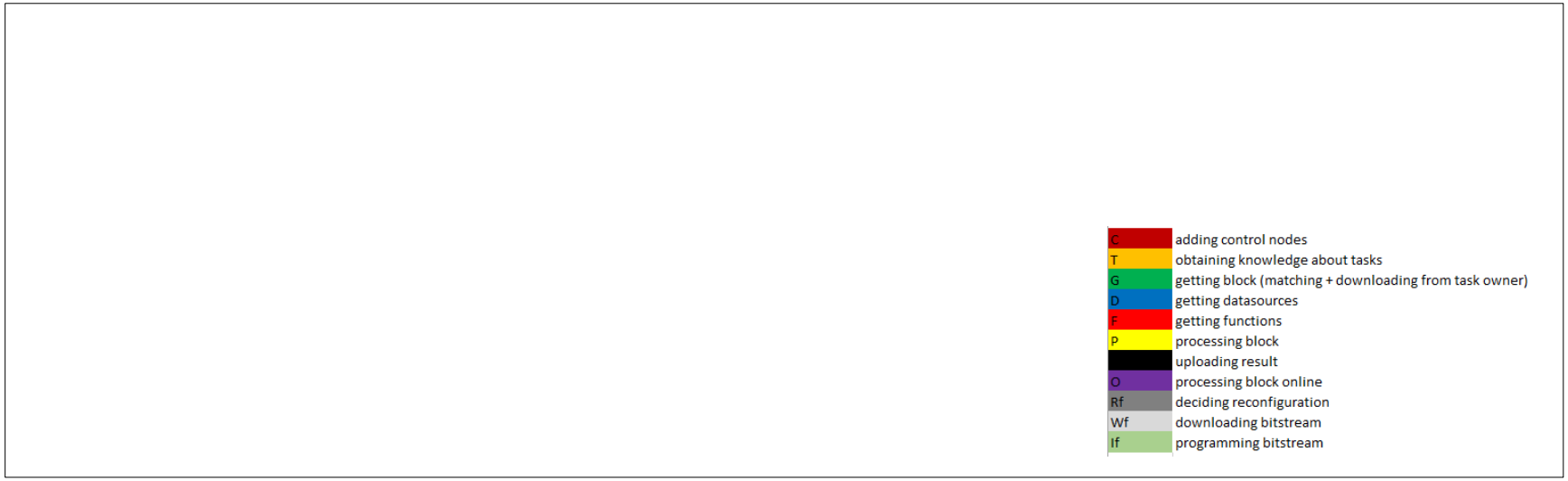


Fig. 75. Gantt graph: system behavior for non-optimized and optimized operations

REFERENCES

[AAS12] A. Al-Wattar, S. Areibi, and F. Saffih, *Efficient On-line Hardware/Software Task Scheduling for Dynamic Run-time Reconfigurable Systems*, Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, Shanghai, 2012, pp. 401-406. doi: 10.1109/IPDPSW.2012.50

[AC15] M. Amoretti, S. Cagnoni, Toward Collective Self-Awareness and Self-Expression in Distributed Systems, in Computer, vol. 48, no. 7, pp. 29-36, July 2015, doi: 10.1109/MC.2015.208

[And04] D. Anderson, *BOINC*: *A System for Public-Resource Computing and Storage*, Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, 2004, s. 4–10.

[Ban00] P. Banerjee et al., *A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems*, Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on, Napa Valley, CA, 2000, pp. 39-48. doi: 10.1109/FPGA.2000.903391

[Boi09] http://www.boincstats.com/

[BP02] K. Bondalapati, and V. Prasanna, *Reconfigurable computing systems*, In Proceedings of the IEEE, vol. 90, no. 7, pp. 1201-1217, Jul 2002. doi: 10.1109/JPROC.2002.801446

[BTL10] B. Betkaoui, D. Thomas, and W. Luk, *Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing*, Field-Programmable Technology (FPT), 2010 International Conference on, Beijing, 2010, pp. 94-101. doi: 10.1109/FPT.2010.5681761

[Buy02] R. Buyya, *Economic-based Distributed Resource Management and Scheduling for Grid Computing*, Doctoral Dissertation, School of Computer Science and Software Engineering, Monasch University, Melbourne, 2002.

[CGP13] D. Cemin, M. Götz, and C. Pereira, *Dynamically reconfigurable hardware/software mobile agents*, Design Automation for Embedded Systems, vol. 18, no. 1, 2013, pp. 39-60, doi=10.1007/s10617-013-9116-3

[Chm10] G. Chmaj, *Optimization of data flows in public distributed computation systems*, Doctoral Dissertation, Wroclaw University of Technology, 2010.

[CL13] G. Chmaj, and S. Latifi, *Decentralization of A Multi Data Source Distributed Processing System Using A Distributed Hash Table*, International Journal of Communications, Network and System Sciences, vol. 6, no. 10, pp. 451-458, 2013, doi:10.4236/ijcns.2013.610047

[CS15] G. Chmaj, and H. Selvaraj, *Distributed processing applications for Unmanned Aerial Vehicles: a survey*, 23rd International Conference on Systems Engineering (ICSEng 2014), Las Vegas, USA, 2014, Progress in Systems Engineering: Advances in Intelligent Systems and Computing, vol. 1089, 2015, pp 449-454

[CS16] G. Chmaj, and H. Selvaraj, *Energy-Efficient Computing Solutions for Internet of Things with ZigBee Reconfigurable Devices*, IJSI vol. 4, no. 1, 2016, pp. 31-47, 2016. doi:10.4018/IJSI.2016010103

[CSG13] G. Chmaj, H. Selvaraj, and L. Gewali, *Tracker-node model for energy consumption in reconfigurable processing systems*, XVIII International Conference on Systems Science, Wroclaw, Poland, pp. 503-512, 2013, doi: 10.1007/978-3-319-01857-7_49

[CZE12] G. Chmaj, D. Zydek, Y. Elhalwagy, and H. Selvaraj, *Overlay-NoC and H-Phy based computing using Modern Chip MultiProcessors*, 2012 IEEE International Conference on Electro Information Technology, Indianapolis, 2012, pp. 1-6, doi:10.1109/EIT.2012.6220604

[CW12] G. Chmaj, and K. Walkowiak, *Decision Strategies for a P2P Computing System*, Journal of Universal Computer Science, vol. 18, no. 5, pp. 599-622, 2012, doi: 10.3217/jucs-018-05-0599

[CWT12] G. Chmaj, K. Walkowiak, M. Tarnawski, and M. Kucharzak, *Heuristic Algorithms for Optimization of Task Allocation and Result Distribution in Peer-to-Peer Computing Systems*, International Journal of Applied Mathematics and Computer Science, vol. 22, no. 3, pp. 733-748, 2012, doi:10.2478/v10006-012-0055-0

[Deh04] A. DeHon et al., Design patterns for reconfigurable computing, Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on, 2004, pp. 13-23. doi: 10.1109/FCCM.2004.29

[DMI11] T. Delot, N. Mitton, S. Ilarri, and T. Hien, *Decentralized pull-based information gathering in vehicular networks using GeoVanet*, Proceedings 12th International Conference on Mobile Data Management, Lulea, 2011, pp. 174-183.

[DoD13] Department of Defense, Unmanned Systems Integrated Roadmap FY2013-2038, 2013

[FBP08] E. Freitas, A. Binotto, C. Pereira, A. Stork, and T. Larsson, *Dynamic Reconfiguration of Tasks Applied to an UAV System Using Aspect Orientation*, Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on, Sydney, NSW, 2008, pp. 292-300. doi: 10.1109/ISPA.2008.69

[FC05] W. Fu and K. Compton, An execution environment for reconfigurable computing, Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on, 2005, pp. 149-158. doi:10.1109/FCCM.2005.19

[GBA12] D. Gachet, M. de Buenaga, F. Aparicio, and V. Padrón, *Integrating Internet of Things and Cloud Computing for Health Services Provisioning: The Virtual Cloud Carer Project*, Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on, Palermo, 2012, pp. 918-921. doi: 10.1109/IMIS.2012.25

[HWH13] C. M. Hsieh, Z. Wang, and J. Henkel, *DANCE: Distributed application-aware node configuration engine in shared reconfigurable sensor networks*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, Grenoble, France, 2013, pp. 839-842. doi: 10.7873/DATE.2013.177

[JN09] L. Jóźwiak, N. Nedjah, Modern Architectures For Embedded Reconfigurable Systems A Survey, Journal of Circuits, Systems and Computers, Vol. 18(2009), No. 2, pp. 209-254, doi:10.1142/S0218126609005034

[JNF10] L. Jóźwiak, N. Nedjah, M. Figueroa, Modern development methods and tools for embedded reconfigurable systems: A survey, Integration, the VLSI Journal, vol. 43, no. 1, 2010, pp. 1-33, doi:10.1016/j.vlsi.2009.06.002

[KBM02] K. Krauter, R. Buyya, and M. Maheswaran, *Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing*, International Journal of Software: Practice and Experience (SPE), Vol. 32, No. 2, 2002, s. 135–164.

[KDW10] S. Kestur, J. Davis, and O. Williams, *BLAS Comparison on FPGA, CPU and GPU*, VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on, Lixouri, Kefalonia, 2010, pp. 288-293. doi: 10.1109/ISVLSI.2010.84

[KJ07] D. Kearney, M. Jasiunas, Using Simulated Partial Dynamic Run-Time Reconfiguration to Share Embedded FPGA Compute and Power Resources across a Swarm of Unpiloted Airborne Vehicles, EURASIP Journal on Embedded Systems, vol. 2007, pp. 1–12, doi:10.1155/ES/2007/48521

[KK08] T. Kwok, Y. Kwok, On the Design, Control, and Use of a Reconfigurable Heterogeneous Multi-Core System-on-a-Chip, Proceedings of the 17th International Heterogeneity in Computing Workshop (HCW 2008), in conjunction with the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), Miami, Florida, USA, 2008

[KKG12] A. Khiar, N. Knecht, L. Gantel, S. Lkad, and B. Miramond, *Middleware based executive for embedded reconfigurable platforms*, Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on, Karlsruhe, 2012, pp. 1-6.

[KNP01] K. Kurowski, J. Nabrzyski, and J. Pukacki, *User preference driven multiobjective resource management in grid environments,* Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, Brisbane, Qld., 2001, pp. 114-121. doi: 10.1109/CCGRID.2001.923183

[KPR13] E. Kuhn, M. Prellwitz, M. Rohrer, and J. Sieck, *A distributed middleware for applications of the Internet of Things,* Intelligent Data Acquisition and Advanced

Computing Systems (IDAACS), 2013 IEEE 7th International Conference on, Berlin, 2013, pp. 517-520. doi: 10.1109/IDAACS.2013.6662739

[LZL13] J. Li, Y. Zhou, and L. Lamont, *Communication architectures and protocols for networking unmanned aerial vehicles*, Globecom Workshops (GC Wkshps), 2013 IEEE, Atlanta, GA, 2013, pp. 1415-1420. doi: 10.1109/GLOCOMW.2013.6825193

[NAO12] M. Nadeem, I. Ashraf, S. Ostadzadeh, S. Wong, and K. Bertels, *Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements*, Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, Shanghai, 2012, pp. 79-90. doi: 10.1109/IPDPSW.2012.6

[NON11] M. Nadeem, S. Ostadzadeh, M. Nadeem, S. Wong, and K. Bertels, *A Simulation Framework for Reconfigurable Processors in Large-Scale Distributed Systems*, Parallel Processing Workshops (ICPPW), 2011 40th International Conference on, Taipei City, 2011, pp. 352-360. doi: 10.1109/ICPPW.2011.50

[NW02] H. Naji, and B. E. Wells, *On incorporating multi-agents in combined hardware/software based reconfigurable systems - a general architectural framework,* System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on, 2002, pp. 344-348. doi: 10.1109/SSST.2002.1027064

[NWE04] H. Naji, B. Wells, and L. Etzkorn, *Creating an adaptive embedded system by applying multi-agent techniques to reconfigurable hardware,* Future Generation Computer Systems, vol. 20, 2004. pp. 1055–1081, doi:10.1016/j.future.2004.02.002

[Ola12] R. Olay, *Accelerating Time-to-Market Using an FPGA and Customizable SoC Methodology*, RTC Magazine, 2012

[Ost08] B. Osterloh, H. Michalik, B. Fiethe and K. Kotarowski, SoCWire: A Network-on-Chip Approach for Reconfigurable System-on-Chip Designs in Space Applications, Adaptive Hardware and Systems, 2008. AHS '08. NASA/ESA Conference on, Noordwijk, 2008, pp. 51-56. doi: 10.1109/AHS.2008.43

[PTD13] K. Pocek, R. Tessier, A. DeHon, Birth and Adolescence of Reconfigurable Computing: A Survey of the First 20 Years of Field-Programmable Custom Computing Machines, Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on, Seattle, WA, 2013, pp. 1-17. doi: 10.1109/FPGA.2013.6882273

[PZ07] K. Puttaswamy, and B. Zhao, *A Case for Unstructured Distributed Hash Tables,* Proceedings IEEE Global Internet Symposium, Anchorage, 2007, pp. 7-12. doi: 10.1109/GI.2007.4301423

[PZC14] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, *Context Aware Computing for The Internet of Things: A Survey*. IEEE Communications Surveys & Tutorials, vol. 16, no. 1, 2014, pp. 414-454.

[RHM12] E. Rosas, N. Hidalgo, and M. Marin, *Two-Level Result Caching for Web Search Queries on Structured P2P Networks,* Proceedings IEEE 18th International Conference on Parallel and Distributed Systems, Singapore 2012, pp. 221-228.

[Roz04] B. Rozanski, *Kamelot - protocol for distributed computing with honest online result verification*, Communications, 2004 International Zurich Seminar on, 2004, pp. 164-167. doi: 10.1109/IZS.2004.1287414

[SAH13] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou, and C. Lung, *Smart Home: Integrating Internet of Things with Web Services and Cloud Computing*, Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, Bristol, 2013, pp. 317-320. doi: 10.1109/CloudCom.2013.155

[SCP02] R. Scrofano, S. Choi, and V. Prasanna, *Energy efficiency of FPGAs and programmable processors for matrix multiplication*, Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on, 2002, pp. 422-425. doi: 10.1109/FPT.2002.1188725

[SG11] I. Sourdis, G. Gaydadjiev, HiPEAC: Upcoming Challenges in Reconfigurable Computing, Reconfigurable Computing: From FPGAs 35 to Hardware/Software Codesign, 2011, doi:10.1007/978-1-4614-0061-5_3

[SKS09] W. Song, S. Kim, S. Seok, and D. Choi, *Pastry based Sensor Data Sharing*, Proceedings of 18th International Conference on Computer Communications and Networks, San Francisco, 2009, pp. 1-6.

[SS09] S. Samara, and G. Schomaker, *Self-Adaptive OS Service Model in Relaxed Resource Distributed Reconfigurable System on Chip (RSoC)*, Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:, Athens, 2009, pp. 1-8. doi: 10.1109/ComputationWorld.2009.104

[SWP04] C. Steiger, H. Walder, M. Platzner, Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks, IEEE Transactions On Computers, Vol. 53, No. 11, November 2004, pp. 1393-1407

[TCC11] J. Timpanaro, T. Cholez, I. Chrisment, and O. Festor, *When KAD meets BitTorrent - Building a Stronger P2P Network*, Proceedings IEEE International Parallel & Distributed Processing Symposium, Shanghai, 2011, pp.1635-1642.

[TH10] H. Takeo, and U. Hiroshi, *Dynamically Reconfigurable Network Nodes in Cloud Computing Systems*, NEC Technical Journal, Vol.5 No.2, 2010

[TMK06] F. Travostino, J. Mambretti, and G. Edwards, *Grid Networks Enabling grids with advanced communication technology*, Wiley, 2006.

[Tom12] T. Tomic, et al., Toward a Fully Autonomous UAV, Research Platform for Indoor and Outdoor Urban Search and Rescue, IEEE Robotics & Automation Magazine, 2012, pp. 46-56, doi:10.1109/MRA.2012.2206473

[TPT12] M. Tol, Z. Pohl, and M. Tichy, *A framework for self-adaptive collaborative computing on reconfigurable platforms*, Advances in Parallel Computing, 579-586, 2012

[UQ05] F. Umer, and A. Qayyum, *Architecture for Decentralized, Distributed Event Communication Mechanism through Overlay Network*, Proceedings IEEE Symposium on Emerging Technologies, Islamabad, 2005, pp. 252-257.

[WSK00] L. Wills, S. Sander, S. Kannan, A. Kahn, J. Prasad, and D. Schrage, *An open control platform for reconfigurable, distributed, hierarchical control systems*, Digital Avionics

Systems Conference, 2000. Proceedings. DASC. The 19th, Philadelphia, PA, 2000, pp. 4D2/1-4D2/8 vol.1. doi: 10.1109/DASC.2000.886955

[Zha04] Z. Zhang, *The power of DHT as a logical space*, Distributed Computing Systems, 2004. FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of, 2004, pp. 325-331. doi: 10.1109/FTDCS.2004.1316634

CURRICULUM VITAE

Grzegorz Chmaj, Ph.D.

4505 S Maryland Pkwy
Department of Electrical and Computer Engineering
University of Nevada, Las Vegas
Las Vegas, NV, 89154
e-mail: grzegorz.chmaj@gmail.com
www: http://faculty.unlv.edu/chmaj

EDUCATION

Wrocław University of Technology, Faculty of Electronics, Wrocław, Poland
Ph.D., Computer Science, June 2010
M.Sc., Computer Science, June 2005

PROFESSIONAL EXPERIENCE

University of Nevada, Las Vegas, Department of Electrical and Computer Engineering
Position: Laboratory Manager (06/2012 – current)
Duties: Classes teaching, designing classes' contents and instructions. Laboratories supervision and management. High performance computational cluster – design and supervision.

University of Nevada, Las Vegas, Department of Electrical and Computer Engineering
Position: Graduate Assistant (08/2011 – 05/2012)
Duties: Teaching assistant, website administrator.

Volvo IT
Position: IT Systems Analyst (02/2011 – 08/2011)
Duties: Analyzing systems logic, requirements management, team and work management.

Wrocław University of Technology
Position: Part Time Instructor (02/2011 – 05/2011)
Duties: Lecturing two graduate level classes.

Wrocław University of Technology
Position: Co-I for two grant programs (10/2009 – 08/2011)
Duties: Creating experimentation systems, scientific research, writing papers, and presentations.

Volvo IT
Position: Software developer / designer (08/2005 – 01/2011)
Duties: Designing new solutions to Volvo IT / Ford IT car warranty system, designing distributed processing solutions for Volvo IT / Ford IT, software development and software systems analysis, employee mentoring, team leading and management, team / work tasks assignment and coordination.

Volvo IT
Position: Data archive / digitizer (01/2005 – 06/2005)
Duties: Input data quality assurance, optimization of data flow, software development, team leading.

AJAN CNC Makilanarı
Position: C++ developer trainee (AIESEC program) (07/2004 – 9/2004)
Duties: Scientific research of nesting problems, software development for nesting industry.

Wrocław University of Technology
Position: Part time instructor / laboratory supervisor (10/2003 – 01/2005)
Duties: Classes instructor, laboratories maintenance, design of research systems, research team supervision.

RESEARCH INTERESTS

- reconfigurable processing systems,
- high performance computing,
- distributed systems,
- distributed computing/processing systems,
- embedded systems,
- grids, public computation architectures,
- effective usage of computing and networking resources,
- parallel processing architectures,
- computer networks: computer network management,
- Peer-to-Peer systems,
- distributed processing applied to unmanned aerial systems (UAS, multi-UAV),
- distributed processing applied to Internet of Things,
- optimization techniques,
- linear programming,
- software development of research simulators.

ACADEMIC WORK

Review work:

- Doctoral Dissertation for University of Technology Sydney
- IEEE Transactions on Vehicular Technology
- Mechanical Systems and Signal Processing (Elsevier)
- Journal of Network and Systems Management
- Journal of Intelligent and Robotic Systems
- Journal of Information Science
- Special issue: CISIS´12- Logic Journal of the IGPL Guest Editors
- Journal of Telecommunications and Information Technology
- Journal of Computer and Communications (JCC)
- International Journal of Communication Networks and Distributed Systems
- Engineering Science and Technology, an International Journal (JESTCH)
- International Journal of Communications, Network and System Sciences (IJCNS)
- Computers Journal (MDPI AG)
- International Conference on Systems Engineering ICSEng 2014
- 48th Hawaii International Conference on System Sciences
- Mediterranean Embedded Computing Resources / EUROMICRO/IEEE Workshop on Embedded and Cyber-Physical Systems 2015

Committees:

- Technical Program Committee Member for INNOV 2015 Conference
- Steering Committee member of ICSEng conference
- Editorial Advisory Board member of International Journal of Electronics and Telecommunications (IJET)
- Program Committee member of Special Issue CISIS´13 - Logic Journal of the IGPL (International Conf. on Computational Intelligence in Security for Information Systems)
- Technical Program Committee member of International Academy, Research, and Industry Association (IARIA)
- Technical Program Committee for The Third International Conference on Communications, Computation, Networks and Technologies
- Technical Program Committee for The Sixth International Conference on Advances in Future Internet
- Special Session on Intelligent Systems and Software Engineering Advances (21st International Conference on Information and Software Technologies (ICIST)): Program Committee Member
- Asia-Pacific Conference on Computer Aided System Engineering (APCASE): Technical Program Committee Member
- Intelligent Systems and Software Engineering Advances 2015 (Lithuania): Program Committee Member
- Polish-British Workshop: Steering Committee Member

SKILLS

- building energy models, research data analysis, discrete simulation, MIP modelling, CPLEX optimization, MIP problems relaxing, building software for research experimentation, building computational environments for the research, building heuristic algorithms based on MIP models, strong reviewing skills, laboratory setup.
- Matlab, Quartus II / DE2 hardware, Microsoft Office, Microsoft Visual Studio, Aptana (+others based on Eclipse), Rocks Cluster Environment (SGE+HPC), CPLEX Optimization Package, Aldec Active HDL, Cadence, Altium Designer, Synopsys.
- Embedded systems (Arduino, BeagleBoard XM, BeagleBone, Raspberry PI, Stellaris EKS-LM3S3748), Laboratory electronic equipment (Emona TIMS-301, function generators, spectrum analyzers etc.), Computer networks (configuration, maintenance).
- Programming languages: ANSI C, C++, .NET/C#, Java (SE/EE/EJB), VHDL, Verilog, SystemC. Script languages: Ruby, Python, Ruby on Rails.
- Databases: SQL, DB/2, mysql, postgresql.
- Operating systems: Microsoft Windows, Linux (Debian, Ubuntu, RedHat, CentOS), Unix, i5/OS, IBM OS/2 Warp 4.5 / eComStation 1.1.