2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Task Scheduling in Large-scale Distributed Systems Utilizing Partial Reconfigurable Processing Elements

M. Faisal Nadeem, Imran Ashraf, S. Arash Ostadzadeh, Stephan Wong, and Koen Bertels
Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
{M.F.Nadeem, I.Ashraf, S.A.Ostadzadeh, J.S.S.M.Wong, K.L.M.Bertels}@TUDelft.nl

***Abstract*—Recent progress in processing speeds, network bandwidths, and middleware technologies have contributed towards novel computing platforms, ranging from large-scale computing clusters to globally distributed systems. Consequently, most current computing systems possess different types of heterogeneous processing resources. Entering into the peta-scale computing era and beyond, reconfigurable processing elements such as Field Programmable Gate Arrays (FPGAs), as well as forthcoming integrated hybrid computing cores, will play a leading role in the design of future distributed systems. Therefore, it is important to develop simulation tools to measure the performance of reconfigurable processors in the current and future distributed systems. In this paper, we propose the design of a simulation framework to investigate the performance of reconfigurable processors in distributed systems. The framework incorporates the partial reconfigurable functionality to the reconfigurable nodes. Depending on the available reconfigurable area, each node is able to execute more than one task simultaneously. Furthermore as a case study, we present a simple task scheduling algorithm to verify the functionality of the simulation framework. The proposed algorithm supports the scheduling of tasks on partially reconfigurable nodes. The simulation results are based on various experiments and they provide a comparison between full (one node-one task mapping) and partial (one node-multiple tasks mapping) configuration of the nodes, for the same set of parameters in each simulation run. Results suggest that the *average wasted area per task* is less as compared to the full configuration, verifying the functionality of the simulation framework.**

***Index Terms*—Simulation framework; Distributed systems; Partial reconfiguration; Reconfigurable resources; Resource management.**

I. Introduction

In recent decades, many new computing platforms have emerged due to research and development in processing resources, network speeds, and middleware services [1][2]. These platforms range from large-scale clusters to globally dispersed grid systems and contain diverse and heterogeneous processing resources of different number and types. These large-scale distributed computing systems are expected to possess millions of cores providing performance in peta-scale [3][4]. The growth in computing power in distributed systems has created new paradigms and opportunities for exploration of novel methods to utilize these resources in effective manners [5]. Nevertheless, this scenario leads to an enormous complexity with many design and optimization alternatives for resource management, application scheduling, and run-time systems.

In the design of next-generation distributed systems and supercomputers, acceleration resources such as FPGAs, Graphics Processing Units (GPUs), and upcoming integrated hybrid computing cores will play a significant role [6][7]. Moreover, GPUs, FPGAs, and multi-cores provide impressive computing alternatives, where some applications allow several orders of magnitude speedup over their General-Purpose Processor (GPP) counterpart. Therefore, future computational systems will utilize these resources to serve as their main processing elements for appropriate applications and in some cases, as co-processors to offload certain compute-intensive parts of applications from the GPPs.

Advances in reconfigurable computing technology (e.g., FPGAs) over the past decade have significantly raised interest in high-performance paradigm [8]. Some of the characteristics of reconfigurable hardware include, configurability, functional flexibility, power efficiency, ease of use, extensibility (adding new functionality), (reasonably) high performance, hardware abstraction, and scalability (by adding more soft-cores). Due to these reasons, multi-FPGA systems are quite an impressive solution for high-performance and distributed computing systems [9][10]. In many research fields of high-performance computing, FPGAs have emerged, either as custom-made signal processors, embedded soft-core processors, multipliers, prototyping designs, systolic arrays, or custom computing architectures. With the emergence of high-level C-like programming environments, FPGAs are continuously improving. However, their design complexity and utilization in a high-performance distributed system are open to innovative research. Various different approaches include efficient resource management, design-space exploration, hardware-software co-design, job scheduling algorithms, reconfigurability, and simulation and synthesis frameworks. With the prospects of including new processing elements (such as FPGAs) in distributed systems, application task scheduling in these systems has become more pertinent and significant beyond its original scope. Similarly, new methodologies like partial run-time reconfiguration for high-performance reconfigurable computing are being exploited [11]. In *partial reconfiguration,* a portion of an FPGA is

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.6

dynamically modified while the remaining regions continue to operate without any disruption. It allows an efficient utilization of FPGA by only configuring the required functionality at any point in time.

In our previous paper [12], we presented the design of a simulation framework for the resource management and tasks scheduling for dynamic reconfigurable processing nodes in a distributed computing system. We termed our proposed simulation framework as **D**ynamic **Re**configurable **A**utonomous **M**any-task **Sim**ulator (DReAMSim). In this work, we have extended our simulation framework to provide the following contributions:

- Design and implementation to incorporate partial reconfigurable functionality to the reconfigurable nodes in DReAMSim. A node region can be reconfigured at runtime while the other regions are already configured and are executing some tasks. In this way, a particular node can accommodate more than one task simultaneously, depending upon its available reconfigurable area.
- Design of efficient data structures to maintain the dynamic statuses of the nodes.
- As a case study, a task scheduling algorithm is proposed and implemented to verify the functionality of the simulation framework. The algorithm supports the scheduling of tasks on partially reconfigurable nodes.

The simulation results are based on various experiments, and they provide a comparison between full (one task per node) and partial (multiple tasks per node) reconfiguration of the nodes, for the same set of parameters in each simulation run. Results suggest that the *average wasted node area per task* in case of partial reconfiguration is less as compared to full reconfiguration, verifying the functionality of the simulation framework.

The remainder of the paper is organized as follows. Section II presents related work and the basic concept. Section III presents the top-level organization of the DReAMSim simulation framework. In Section IV, we detail the formal system model and the design and implementation the DReAMSim. In Section V, we propose a task scheduling algorithm which takes into account the partial reconfigurability of the nodes in a distributed system. The simulation environment and results are discussed in Section VI. Finally, Section VII provides the conclusions and future work.

II. Related Work and Basic Concept

GridSim [13] and SimGrid [14] are the most notable simulation tools for application task scheduling, grid economy, and resource management for distributed computing systems. These tools provide extensive frameworks for modeling of computing resources comprising of GPPs, with fixed computing capacities for every simulation run. Due to this constraint, these simulation tools can not be modified to add reconfigurability of nodes. However, CRGridSim [15] was an effort to extend GridSim to add reconfigurable elements in grid systems, but the proposed extensions were limited. It only included a speedup factor of a reconfigurable element over a GPP, as the primary reconfiguration parameter. Many other significant parameters, such as area utilization, reconfigurability, reconfiguration delay, hardware technology, application model, and application size were not considered. In our previous work [12], we proposed the design of a simulation framework for reconfigurable processors in large-scale distributed systems. A user can model reconfigurable processing elements and implement various scheduling policies. In the current work, we have extended our framework to include the option for partial reconfigurability among the nodes. Hence, a node can be reconfigured for more than one *configurations* at one time.

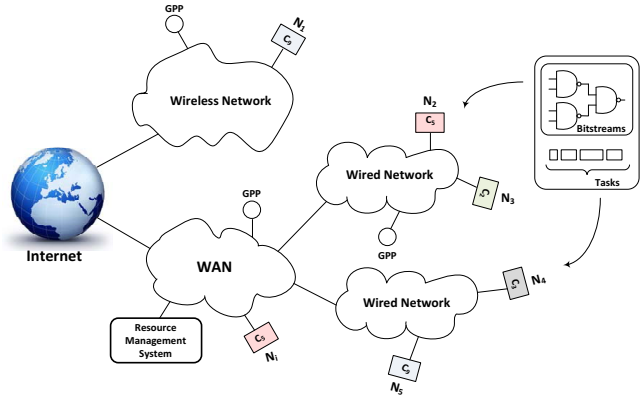


Figure 1: A conceptual overview of a distributed system with reconfigurable nodes.

Figure 1 depicts a conceptual overview of a large-scale distributed system containing reconfigurable processing nodes, as well as GPPs. It consists of a *Resource Management System (RMS)* which handles the monitoring, load distribution, application task scheduling among different nodes in the system. It can be noted that each reconfigurable node contains a particular configuration (denoted as C_i) of a processor of a certain type (P_{type}). It can execute an application task, which requires a preferred processor configuration (denoted as C_{pref}). An existing C_i on a node can be changed by sending a bitstream of a different configuration. The *RMS* distributes the tasks onto suitable nodes specified by some task scheduling algorithm. If a task prefers a certain processor configuration (C_{pref}), then the *RMS* reconfigures a suitable node by sending the corresponding bitstream and sends the task for processing.

III. The DReAMSim Simulation Framework

Figure 2 depicts the top-level organization of the DReAMSim simulation framework. There are 4 different subsystems in the proposed framework; *input*, *information*, *core*, and *output* subsystems. Each subsystem is composed of various modules.

The *input* subsystem: it is the input interface to allow user inputs to the framework. It allows a user to set *application specifications* as well as *user-defined resource specifications*. It generates synthetic tasks which may require a particular processor configuration (C_{pref}) and required estimated time for the execution of tasks. It can also support

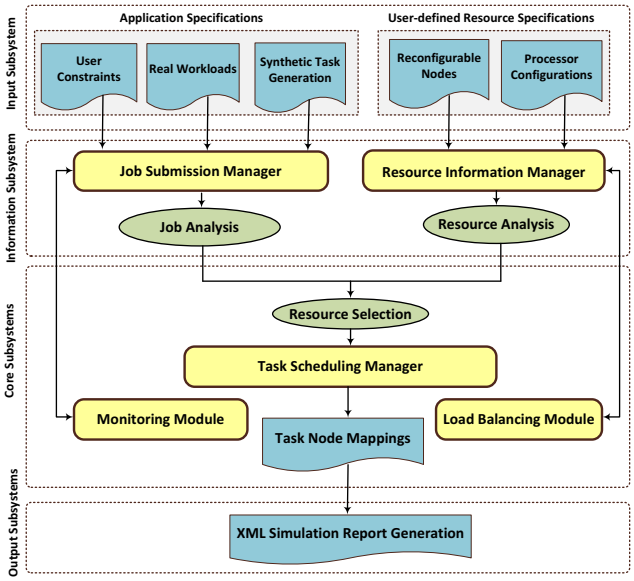


Figure 2: DReAMSim top-level organization.

real workloads and user constraints. A user can specify the task arrival rate and arrival distribution functions. The *User-defined resource specification* module is responsible for generating nodes and different processor *configurations*. It can produce nodes with various reconfigurable area sizes. Reconfiguration method (full or partial reconfiguration) can also be set. For simulation purposes, a user can specify the node upper and lower area limits. For a more realistic study, these limits can be specified according to the real area sizes of the reconfigurable devices available in the market. Similarly, a variety of processor *configurations* can be generated. Again for a realistic experiment, different *configurations* of P_{type} and their corresponding parameters can be specified allowing various processor options for application tasks.

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

The *core* subsystem: this subsystem is the core of the framework and it consists of a *task scheduling manager,* a *monitoring module,* and a *load balancing module.* The *task scheduling manager* can implement different scheduling policies to schedule tasks onto various nodes. Due to the dynamic nature of a distributed system with reconfigurable nodes, the task scheduling process becomes a significant matter. First of all, as a result of the large number of nodes, it is hard to address issues such as load balancing when a task is scheduled to the available set of nodes. Secondly, since the nodes are reconfigurable and the system is dynamic, the scheduling should be adaptive. For these reasons, the framework contains a *load balancing module.* The current states of different nodes can be checked by the *monitoring module.*

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

IV. Design and Implementation

A. System Model

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

$$Node_i\,(TotalArea, AvailableArea, \mathbf{C}, family, caps, state) \quad where\ \ \mathbf{C} = \{C_1, C_2, \cdots, C_m\} \tag{1}$$

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

$$\mathrm{C}_i\,(ReqArea, P_{type}, \mathbf{param}, BSize, ConfigTime) \quad where\ \ \mathbf{param} = \{parameter_1, \cdots, parameter_k\} \tag{2}$$

Where i represents the *configuration number* and $ReqArea$ is the total reconfigurable area required by the configuration i. P_{type} represents the processor configuration required by tasks and **param** is a set of $parameters$ representing a list of attributes of a particular P_{type} processor which provide its architectural details. Some examples of P_{type} are multipliers, systolic arrays, soft-core processors, and custom-made signal processors. One such example is a *parameterizable* soft-core ρ-VEX VLIW processor presented in [16]. It has been implemented on FPGA and can be adopted to several architectural $parameters$. These $parameters$ include the number and types of functional units (multipliers and ALUs), cluster cores, the number of issues, or the number of memory slots. Finally, $BSize$ represents the file size of bitstream for a configuration C_i. Similarly, an application task is defined by the following tuple:

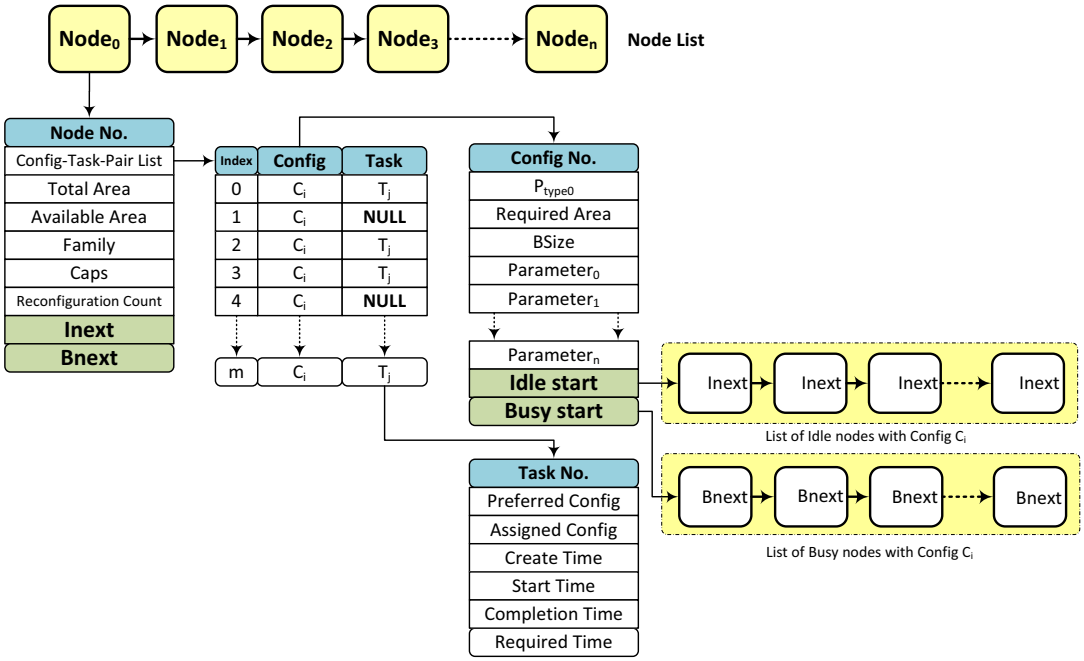


Figure 3: Dynamic data structures in the resource information system.

$$Task_i\,(t_{required}, C_{pref}, data) \tag{3}$$

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

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

$$AvailableArea = \begin{cases} TotalArea & \text{if } m = 0 \\ TotalArea - \sum_{i=1}^{m} ReqArea_i & \text{if } m > 0 \end{cases} \tag{4}$$

Where m represents the cardinality of the set $\mathbf{C}$ in the tuple 1, and it provides the total number of current configurations of the node.

B. Dynamic Data Structures for Resource Management

We designed and implemented effective and dynamic data structures to provide a simple mechanism for the maintenance of information corresponding to dynamically reconfigurable nodes in the *resource information manager*. Figure 3 depicts the proposed data structure. Information regarding all reconfigurable nodes in the system is maintained by a data structure denoted by *node list*. Each item (node) in this list updates the information about a node and it contains a *config-task-pair list*, $TotalArea$, $AvailableArea$, and other node attributes. It also contains two pointers namely, *Inext* and *Bnext*. As depicted in the Figure 3, these pointers are used to link all the idle or busy nodes of a certain configuration. The *config-task-pair list* is another data structure which maintains the *configuration-task* pairs on a particular node. It keeps track of all the configurations on a node. Whenever a new configuration is made on the node, a new *configuration-task* entry is created in the list. In this figure, all the actively running tasks are represented by T_j. If there is no currently running task on a particular configuration on the node, then it is represented by *NULL*. The maximum number of *configuration-task* pairs on a node depends on the $AvailableArea$ of the node.

Each *configuration* instantiates a typical processor configuration of type P_{type}, its $ReqArea$, and a list of its **param**. Additionally, it consists of two pointers *Idle start* and *Busy start*. *Idle start* (or *Busy start*) points to the first node of the linked list of the idle nodes (or busy nodes) configured with this particular *configuration*.

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

C. DReAMSim Class Design

A class diagram of the DReAMSim framework, represented by using simple, unified modeling language (UML) notation, is depicted in the Figure 4. It provides details of different classes implemented in order to generate tasks, nodes, configurations, scheduler, lists, and simulation environment. The framework implements the following important classes:

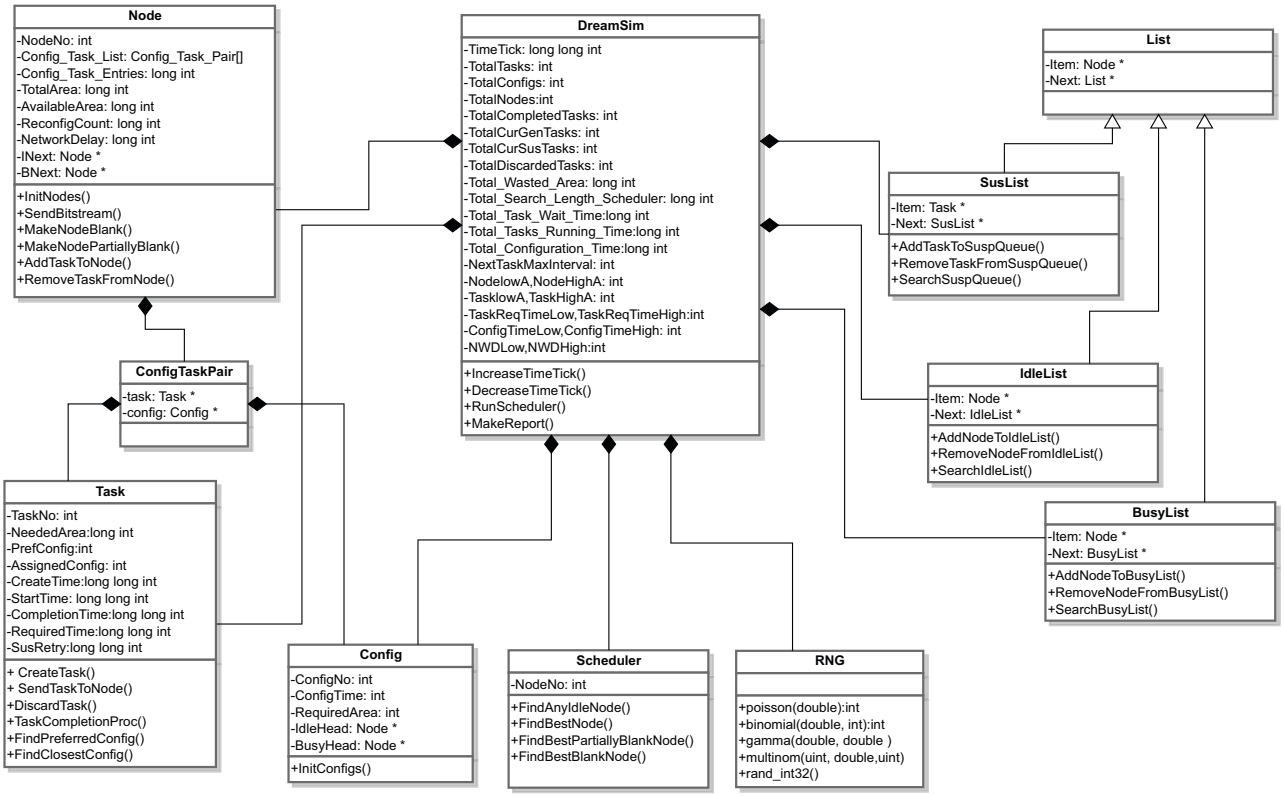


Figure 4: UML model of DReAMSim

- Node: implements a typical reconfigurable node and its capability is defined by $TotalArea$, $AvailableArea$, $family$, and $caps$. Some important methods of this class are described in the following:
  - *InitNodes():* initializes the number of nodes defined by the user, and allots a $TotalArea$ to each node within the upper (*NodeUpperA*) and lower (*NodeLowerA*) area limits.
  - *SendBitstream():* adds a configuration on the node, adjusts the $AvailableArea$ according to the $ReqArea$ of the new configuration, and increases the *reconfiguration count* on the node.
  - *MakeNodeBlank():* removes all the configurations on a particular node, and equates the $AvailableArea$ to the $TotalArea$.
  - *MakeNodePartiallyBlank():* removes one or more existing configurations on a particular node, and readjusts the $AvailableArea$.
  - *AddTaskToNode():* adds a task to a particular node, if it contains the required configuration of the task.
  - *RemoveTaskFromNode():* removes a task from a particular node.
- Task: this is used to represent a task, and is characterized by $t_{required}$,C_{pref}, and $data$. It contains the following methods:
  - *CreateTask():* creates a new task and its attributes, such as t_{create}, $t_{required}$, and its C_{pref}.
  - *SendTaskToNode():* sends a task to a particular node, and calculates its *start time (t_{start}), completion time ($t_{completion}$), and waiting time (t_{wait}).*
  - *TaskCompletionProc():* it is invoked by the scheduler, after a task finishes executing on a particular node. It releases the node to make it available for succeeding tasks. It also updates the corresponding idle and busy lists. Additionally, it reports some statistics after the completion of the task.
  - *FindPreferredConfig():* searches for the C_{pref} of a particular task among all the configurations in the *configurations list*. Currently, a simple linear search is employed but it can be made more intelligent.
  - *FindClosestConfig():* searches for the $C_{ClosestMatch}$ of a particular task if its C_{pref} is not available in the *configurations list*. The criteria for the $C_{ClosestMatch}$ is that its $ReqArea$ is the minimum among all configurations with a $ReqArea$ more than the $ReqArea$ of the C_{pref}.
- Config: this class represents a typical configurations with attributes such as, its $ReqArea$, P_{type}, $parameters$, $BSize$, and $ConfigTime$. It consists of the following method:
  - *InitiConfig():* initializes the *configurations list* and assigns $ReqArea$ and $ConfigTime$ to all the configurations within a user-defined range.
- ConfigTaskPair: it is a dynamic data structure which is

used to keep track of all tasks and their corresponding configurations on a certain node.

- IdleList and BusyList: these classes maintain linked lists for managing information corresponding to idle nodes with a particular configuration. If a task finishes execution on a certain node, then it is added to the idle list of nodes (and removed from the busy list of nodes) with the configuration required by the task. These classes contain the following important methods:
  - *AddNodeToIdleList()* and *AddNodeToBusyList():* adds a node to the idle (or busy) list of nodes with a particular configuration.
  - *RemoveNodeFromIdleList()* and *RemoveNodeFromBusyList():* removes a node from the idle (or busy) list of nodes with a particular configuration.
  - *SearchIdleList()* and *SearchBusyList():* used to traverse the idle (or busy) list of a particular configuration.
- Scheduler: implements various task scheduling policies defined by the user and contains the following methods:
  - *FindBestNode()*: it is invoked to search the *best-node* match for a given task, among all the idle nodes configured with its C_{pref}. The criteria for the *best-match* depends on the scheduling strategies. For instance, *best-match* can be the node which possesses the minimum $AvailableArea$.
  - *FindAnyIdleNode():* search for any idle nodes configured with a configuration other than the C_{pref} of the task. It is explained in the Algorithm 1.
- DreamSim: it is the core class of the simulator and it interacts with other classes to prepare the system to run the user-defined simulations. Two important methods of this class are given in the following:
  - *RunScheduler():* it simulates a certain task scheduling policy implemented in the scheduler.
  - *MakeReport():* accumulates the statistical data during the simulations.
- SusList: this class implements the suspension queue data structure to hold suspended tasks and contains the following methods:
  - *AddTaskToSusQueue():* implements a simple queue data structure which holds the tasks, put in suspension by the scheduler.
  - *RemoveTaskFromSusQueue():* each time a node finishes executing a task, the suspension queue is checked using this method to determine if a suitable task is waiting in the queue which can be executed on the node. It removes the task from the suspension queue and sends it the node.
  - *SearchSusQueue():* traverses the suspension queue to search a particular task.
- RNG: it is a random number generator class which is based on the *Ziggurat Method* [17] using the algorithm described in [18] for generating *Gamma variables*. It provides several random number distributions, such as *Poisson*, *Binomial*, *Gamma*, *Uniform random*, etc. The simulator uses these random number distributions to generate configurations, nodes, and task arrivals.

The method *MakeReport()* in the DReAMSim class accumulates the statistical data at the end of each simulation. Table I provides some important statistics generated by the simulator. *Total scheduler workload* is the sum of *search steps* taken by the simulator to schedule a task and to do different house keeping activities, for instance, updating the idle, busy, and suspension queue lists. A *search step* is a basic unit of exploration to search a memory location. A scheduling step counter *SL* is incremented after each search step that the *scheduler* takes to schedule a task. The methods *IncreaseTimeTick()* and *DecreaseTimeTick()* simulate the progress of time in terms of *timeticks* in a general manner, where *timetick* is a unit of time on a target system. In this respect, the total simulation time is calculated as follows:

$$Total\ simulation\ time = Total\ number\ of\ timeticks \quad (5)$$

The *total wasted area* at any given time is calculated as follows:

$$Total\ wasted\ area = \sum_{i=1}^{N'} AvailableArea_i \quad (6)$$

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

$$Avg\ wasted\ area\ per\ task = Total\ wasted\ area / Total\ tasks \quad (7)$$

The *total waiting time for each task*, t_{wait} is calculated as follows:

$$t_{wait} = t_{start} - t_{create} + t_{comm} + t_{config} \quad (8)$$

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

$$Avg\ waiting\ time\ per\ task = \sum_{j=1}^{T} (t_{wait})_j / Total\ tasks \quad (9)$$

Similarly, we calculate the *total configuration time* as follows:

$$Totalconfigtime = \sum_{k=1}^{C} (ReconfigCount)_k . (ConfigTime)_k \quad (10)$$

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

Table I: Some important DReAMSim performance metrics.

Performance Metric	Description
Average wasted area per task	Average reconfigurable area wasted during the scheduling of all tasks during a simulation run. See Equation 7.
Average running time of each task	Average time lapse from the arrival of the task to the system until its completion.
Average reconfiguration count per node	Average number of reconfigurations performed by each node during the simulation.
Average reconfiguration time per task	Average reconfiguration time required per task during the simulation. See Equation 10.
Average waiting time per task	Average time elapsed from the time a task is submitted to the system until the time it is assigned to a node. See Equation 9.
Average scheduling steps per task	Total number of search links explored by the scheduling system to assign a task to a proper node. This metric closely reflects the quantitative value of the time taken by the scheduling system to accommodate a task.
Total discarded tasks	Total number of tasks discarded during the simulation.
Total scheduler workload	Total number of *search steps* taken by the *resource information module* to perform various housekeeping jobs such as, maintaining the current states of nodes and configurations.
Total used nodes	Total number of nodes utilized during a simulation run.
Total simulation time	Total simulation time required to execute all tasks during the simulation. See Equation 5.

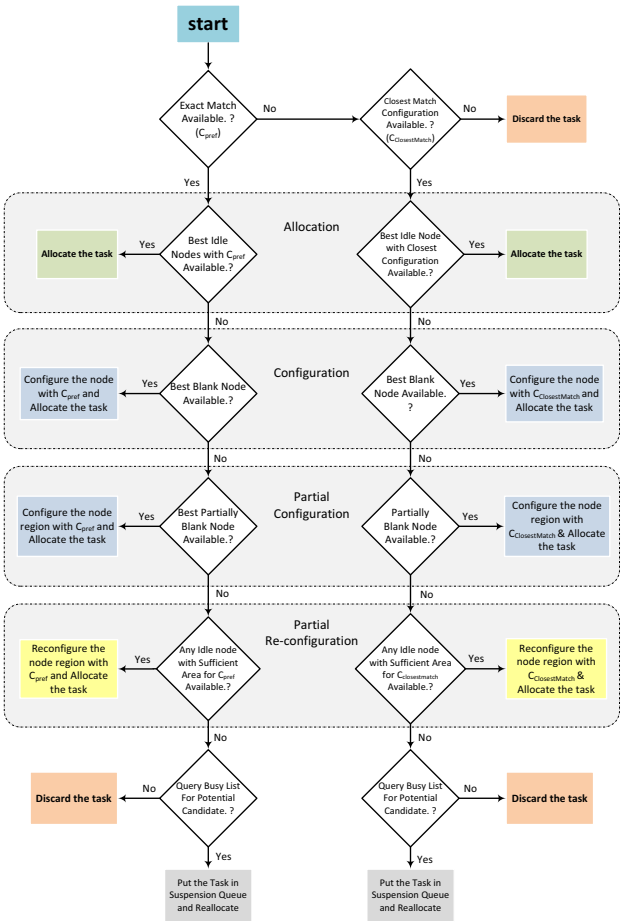


Figure 5: The scheduling algorithm with support for partial reconfigurability of nodes.

V. Task Scheduling Algorithm- A Case Study

As a case study, we tested our framework by implementing a simple dynamic scheduling algorithm, which takes into account the partial reconfigurability of the nodes. It was implemented in the scheduler part of the framework. The algorithmic process is mainly divided into four different parts, as depicted in the Figure 5. Each incoming task is allocated to a particular node by using one of these algorithmic parts namely, *allocation*, *configuration*, *partial configuration*, *partial re-configuration*. Initially, the scheduler decides whether the *exact-match* configuration (or C_{pref}) of the *task* is available in the *configurations list*. If the C_{pref} of the *task* is not available, then the algorithm searches for a *closest-match* configuration (or $C_{ClosestMatch}$) of the *task* in the *configurations list*. However, if $C_{ClosestMatch}$ is also not available, the *task* is discarded. We explain each part of the algorithmic process in the following:

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

Configuration: If the direct *allocation* is not possible due to the absence of idle node(s), then one of the blank nodes is *configured* with the C_{pref} (or $C_{ClosestMatch}$) and the *task* is allocated. A *blank node* is defined as a node with no current configuration.

Partial configuration: If *allocation* or *configuration* can not be performed, the scheduler searches for a node which contains a reconfigurable region with sufficient area to configure the C_{pref} (or $C_{ClosestMatch}$) of the *task*. In this case, the scheduler chooses a node with minimum sufficient region and configures it with the C_{pref} (or $C_{ClosestMatch}$) and allocates the *task*. If there are no nodes with sufficient area are not available, then the scheduler performs *partial re-configuration*, as explained below.

Partial re-configuration: In this case, the scheduler explores any idle nodes which are currently configured with any configuration, other than the C_{pref} (or $C_{ClosestMatch}$) of the *task*. This phase is performed by the *FindAnyIdleNode* method in the scheduler, which is presented by the algorithm 1. It takes the *task* as an input and returns a particular node which can be *re-configured* for the scheduling of the task. The method explores the *entries* of the idle configurations

Algorithm 1 The FindAnyIdleNode Alogrithm

Require: $task$ to be scheduled
  $nodeArea \leftarrow 0$
  $Entries \leftarrow NULL$
  for all $node \in NodeList$ **do**
    $accumIdleArea \leftarrow node.AvailableArea$
    for all $entry \in node.ConfigTaskList$ **do**
      $SearchLength \leftarrow SearchLength + 1$
      $TotalSimWorkLoad \leftarrow TotalSimWorkLoad + 1$
      if $entry$ is idle **then**
        $configArea \leftarrow entry.config.RequiredArea$
        $accumIdleArea \leftarrow accumIdleArea + configArea$
        $Entries \leftarrow entry$
        if $accumIdleArea \geq task.ReqArea$ **then**
          return $node$
        end if
      end if
    end for
  end for
  return $NULL$

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

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

VI. Simulation Environment and Results

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

Simulation parameters: Table II presents various simulation parameters and their values, used in our experiments. Total number of nodes and configurations are set as 200 and 50, respectively. The task arrival interval is set between [1..50] time-ticks with uniform distribution. The total number of tasks in each experiment varies between [1000...100,000] and their $t_{required}$ changes between [100...100,000] time-ticks randomly. Furthermore, the $TotalArea$ of each node ranges between 1000 to 4000 area units (e.g., area slices). Similarly, all the *configurations* require a reconfiguration area

Table II: Different simulation parameter values.

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

($ReqArea$) which is set between 200 to 2000 area units. For 15% of the total tasks, we assign a preferred configuration (C_{pref}) that can not be found in *configurations list*. Therefore, these tasks are assigned to the nodes with $C_{ClosestMatch}$ by the scheduler, dynamically.

We conducted experiments which are based on two different scenarios. In one scenario (*without partial configuration*), the nodes can be configured for only one *configuration* at one time. As a result, each node can execute only one task at one time. In the 2nd scenario (*with partial configuration*), each node is able to accommodate as many configurations as possible, depending upon its $AvailableArea$.

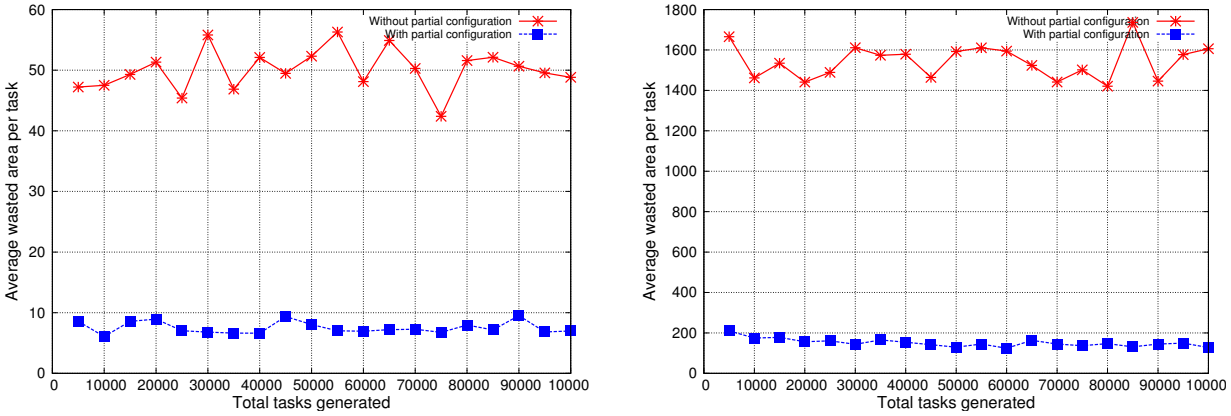
A. Results Discussion

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

The average wasted area per task: Figures 6a and 6b depict the *average wasted area per task* results against a set of tasks varying between $[1000 \ldots 100,000]$ for 100 and 200 nodes, respectively. All the other parameters are set according to the values in Table II. First, it can be noticed from both figures that the *average wasted area per task* is less for the scenario *with partial reconfiguration*. This is due to a possibility of adding more tasks on an already operative node, if it contains sufficient $AvailableArea$ to accommodate the incoming task. In case of the scenario *without partial reconfiguration*, only one task can be executed at one time. As a result, when the node is reconfigured with the C_{pref} (or $C_{ClosestMatch}$) of the task, the remaining area is wasted and can not be utilized for another incoming task until the node finishes executing the current task. Secondly, the quantitative values of *average wasted area per task* for 100 nodes are far less (10-50 area units) as compared to 200 nodes (200-1600 area units). The reason is that the scheduler has a choice of more number of nodes (200 nodes) for each incoming task. As a result, the total accumulated wasted area is more.

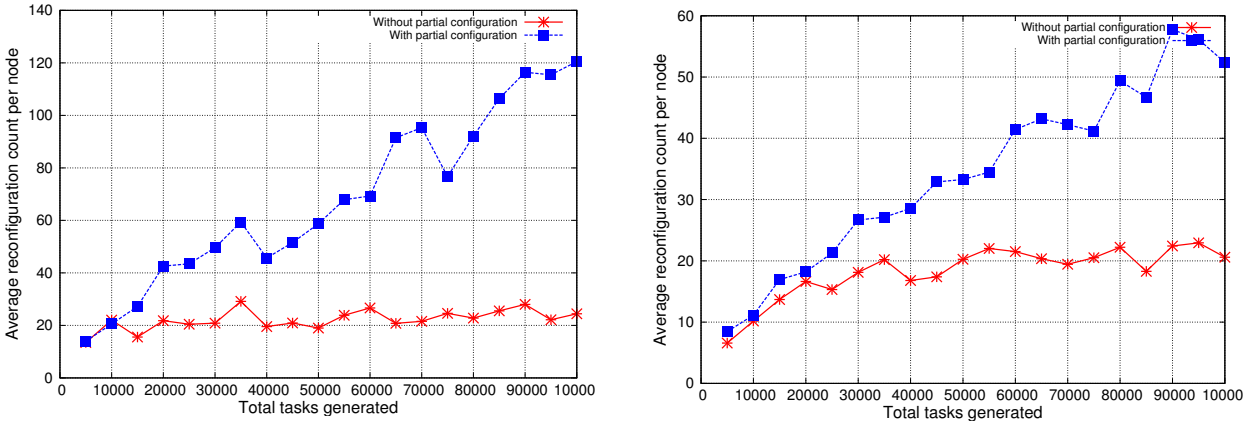
For a fixed set of parameters and 100 nodes, it is expected that the waiting time of each task will be high as compared to 200 nodes. Similarly, it is expected that fewer number of nodes (100 nodes) will be reconfigured more. These results are explained in the following.

The average reconfiguration count per node: Figures 7a and 7b depict that, *with partial reconfiguration*, a typical node



(a) total nodes=100. (b) total nodes=200.

Figure 6: *Average wasted area per task* results for (a) 100 nodes and (b) 200 nodes.



(a) total nodes=100. (b) total nodes=200.

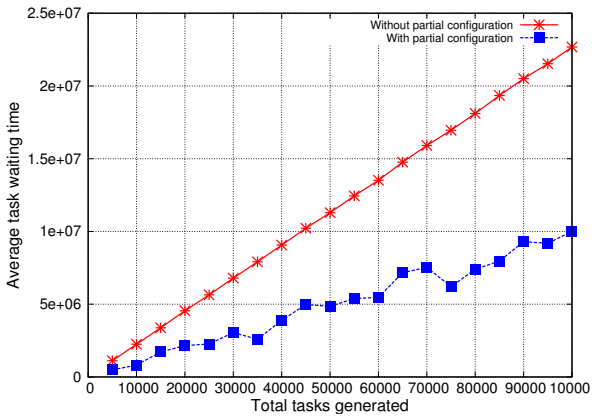
Figure 7: *Average reconfiguration count per task* results for (a) 100 nodes and (b) 200 nodes.

is reconfigured more times (on average) due to more options for the scheduler to assign a task to a node. Hence, the *average reconfiguration count per node* is high as compared to the scenario when only one *configuration* (or one task) is allowed at one time. In case of 100 nodes, as depicted in the Figure 7a, the reconfiguration count is high. It is because the scheduler has less options to schedule each incoming task. It explores any idle nodes which are currently configured with any configuration, other than the C_{pref} (or $C_{ClosestMatch}$) of the *task. As a result,* it reconfigures those idle nodes with C_{pref} (or $C_{ClosestMatch}$) and schedules the task. Hence, the *average reconfiguration count per node* is high.

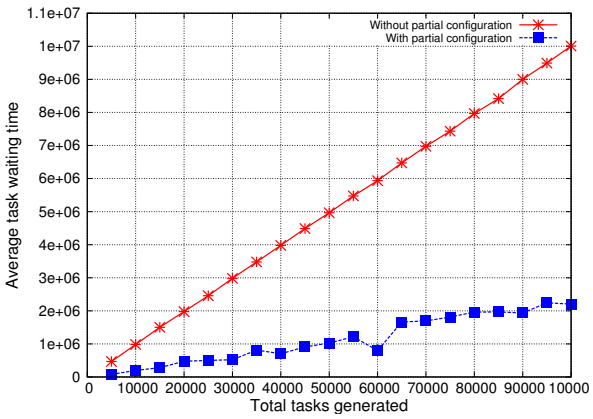
The average waiting time per task: Figures 8a and 8b depict the *average waiting time per task* results for 100 and 200 nodes, respectively. In the scenario *with partial reconfiguration*, the scheduler can immediately send a task to a particular node if it has sufficient $AvailableArea$ to configure its C_{pref} (or $C_{ClosestMatch}$). Therefore, each incoming task requires less waiting time before it is scheduled to a suitable node. In a scenario *without partial reconfiguration*, the scheduler has no options to schedule multiple tasks on a single node, so the *average task waiting times* are much higher. In case of 100 nodes, as depicted in the Figure 8a, the *average waiting time per task* is very high due to a fewer number of nodes. Moreover, the tasks are arriving after a very small interval of [1...50] time-ticks.

The average scheduling steps per task: The *average scheduling steps per task* result comparison between the two scenarios for 200 nodes, is depicted in Figure 9a. In the first case *with partial reconfiguration*, the scheduler can even search for a node *region* to map a task, which reduces the scheduling effort to accommodate a task. This results in less scheduling steps as compared to the case *without partial reconfiguration.*

The total scheduler workload: Figure 9b depicts that the simulator requires more workload in the scenario *without partial reconfiguration.* This is because, the possibilities to schedule a task are limited and more housekeeping is re-

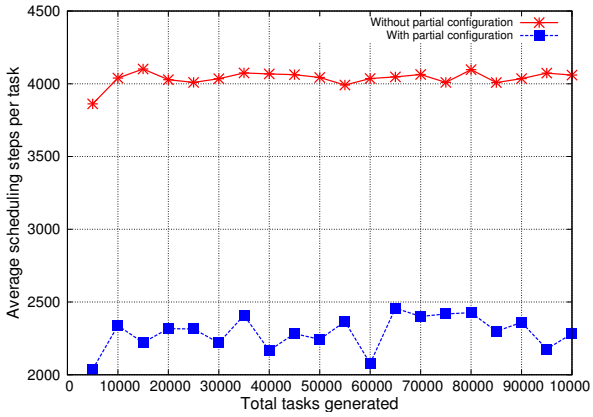


(a) total nodes=100.

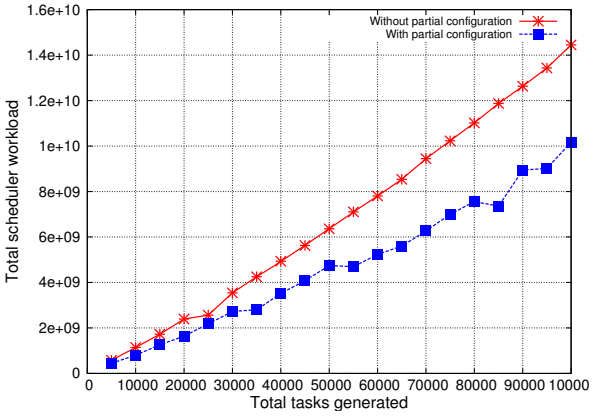


(b) total nodes=200

Figure 8: *Average waiting time per task* results for (a) 100 nodes and (b) 200 nodes.



(a) *Average scheduling steps per task* (total nodes = 200).



(b) *Total simulation workload* (total nodes=200).

Figure 9: *Average scheduling steps per task* and *total simulation workload* results for 200 nodes.

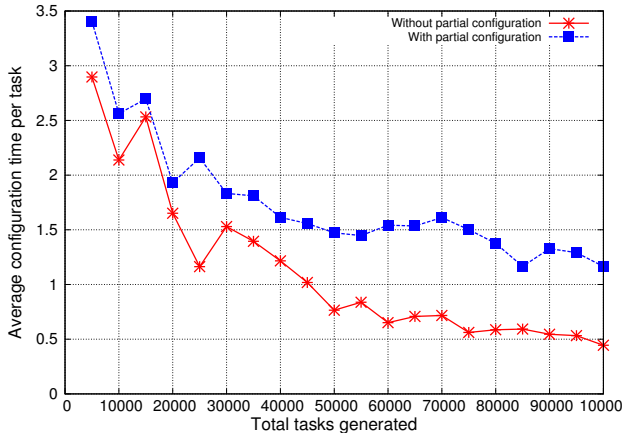


Figure 10: *Average configuration time per task* (nodes=200).

quired.

The average configuration time per task: Since the *average reconfiguration count per node* is much higher in scenario *with partial reconfiguration*, so the *average configuration time per task* is also higher as depicted in Figure 10.

Overall, it can be concluded that for a given set of parameters and *with partial reconfiguration*, the system utilizes less area, less scheduling steps, and scheduler workload. On the other hand, *without partial reconfiguration*, the scheduler has to wait to schedule a task due to an absence of options to accommodate multiple tasks on a node simultaneously.

From the above set of experiments, it can be noted that the scheduler's behavior depends on the reconfiguration methods. Various metrics such as the *average scheduling steps per task*, the *average waiting time per task*, and the *average reconfiguration count per node* follow an expected behavior for the two scenarios explained above.

VII. Conclusions and Future Work

In this paper, we extended our simulation framework to add partial reconfigurable functionality to the reconfigurable nodes in a large-scale computing system. With this extension, each node in the system is able to execute more than one application task simultaneously, depending on the available node area. Based on a proposed dynamic scheduling algorithm, we presented and discussed simulation results obtained from our framework. Our results suggest that the *average wasted area per task* in case of partial reconfiguration scenario, is less as compared to full configuration. Hence, the node utilization can be improved by adding more than one reconfigurations on a node. The proposed simulation framework can be used to test different scheduling policies for a given set of parameters, such as tasks, nodes, configurations, and area bounds, etc. In future work, we will implement load balancing manager to perform a better load distribution among all the nodes. We will implement scheduling policies to schedule task graphs on the distributed system with reconfigurable nodes. We will test the simulation framework with real workloads and realistic scenarios.

References

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

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

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

[4] TeraGrid, "FPGA Resources in Purdue University." http://www.rcac.purdue.edu/teragrid/userinfo/fpga.

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

[6] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, pp. 115–124, 2010.

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

[8] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computer Survey*, vol. 34, no. 2, pp. 171–210, 2002.

[9] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A High-End Reconfigurable Computing System," *Design and test of computers, IEEE*, vol. 22, no. 2, pp. 114–125, 2005.

[10] Baxter, et. al, "Maxwell- a 64 FPGA Supercomputer," in *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 287–294, 2007.

[11] E. El-Araby and I. Gonzalez and T. El-Ghazawi, "Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing," *ACM Transactions in reconfigurable technology systems*, vol. 1, no. 4, pp. 1–23, 2009.

[12] M. F. Nadeem et. al, "A simulation framework for reconfigurable processors in large-scale distributed systems," in *Proceedings of the 2011 40th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 352–360, 2011.

[13] R. Buya and M. M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1175–1220, 2002.

[14] H. Casanova, "Simgrid: a Toolkit for the Simulation of Application Scheduling," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pp. 430–437, 2001.

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

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

[17] G. Marsaglia and W. W. Tsang, "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, vol. 5, no. 8, pp. 1–7, 2000.

[18] G. Marsaglia and W. W. Tsang, "A Simple Method for Generating Gamma Variables," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 363–372, 2000.