Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2016, Article ID 9012909, 24 pages
http://dx.doi.org/10.1155/2016/9012909

Research Article

An Efficient Evolutionary Task Scheduling/Binding Framework for Reconfigurable Systems

A. Al-Wattar, S. Areibi, and G. Grewal

School of Engineering and Computer Science, University of Guelph, Guelph, ON, Canada N1G 2W1

Correspondence should be addressed to S. Areibi; sareibi@uoguelph.ca

Received 13 September 2015; Revised 26 November 2015; Accepted 16 December 2015

Academic Editor: Nadia Nedjah

Several embedded application domains for reconfigurable systems tend to combine frequent changes with high performance demands of their workloads such as image processing, wearable computing, and network processors. Time multiplexing of reconfigurable hardware resources raises a number of new issues, ranging from run-time systems to complex programming models that usually form a reconfigurable operating system (ROS). In this paper, an efficient ROS framework that aids the designer from the early design stages all the way to the actual hardware implementation is proposed and implemented. An efficient reconfigurable platform is implemented along with novel placement/scheduling algorithms. The proposed algorithms tend to reuse hardware tasks to reduce reconfiguration overhead, migrate tasks between software and hardware to efficiently utilize resources, and reduce computation time. A supporting framework for efficient mapping of execution units to task graphs in a run-time reconfigurable system is also designed. The framework utilizes an Island Based Genetic Algorithm flow that optimizes several objectives including performance, area, and power consumption. The proposed Island Based GA framework achieves on average 55.2% improvement over a single-GA implementation and an 80.7% improvement over a baseline random allocation and binding approach.

1. Introduction

In the area of computer architecture choices span a wide spectrum, with *Application Specific Integrated Circuits (ASICs)* and *General-Purpose Processors (GPPs)* being at opposite ends. General-Purpose Processors are flexible, but unlike ASICs, they are not optimized for specific applications. Reconfigurable architectures, in general, and *Field Programmable Gate Arrays (FPGAs)*, in particular, fill the gap between these two extremes by achieving both the high performance of ASICs and the flexibility of GPPs. However, FPGAs are still not a match for the lower power consumed by ASICs, nor for the performance achieved by the latter. One important feature of FPGAs is their ability to be partially reprogrammed during the execution of an application. This Run-Time Reconfiguration capability provides common benefits when adapting hardware algorithms during system run-time including sharing hardware resources to reduce device count, minimizing power, and shortening reconfiguration time [1].

Many embedded application domains for reconfigurable systems require frequent changes to support the high performance demands of their workloads. For example, in telecommunication applications, several wireless standards and technologies, such as WiMax, WLAN, GSM, and WCDMA, have to be utilized and supported. However, it is unlikely that these protocols will be used simultaneously. Accordingly, it is possible to dynamically load only the architecture that is needed onto the FPGA. Also, in an *Unmanned Aerial Vehicle (UAV)*, employing different machine vision algorithms and utilizing the most appropriate based on the environment or perhaps the need to lower power consumption is, yet, another example.

Time multiplexing of reconfigurable hardware resources raises a number of new issues, ranging from run-time systems to complex programming models that usually form a *Reconfigurable Hardware Operating System (ROS)*. The ROS performs online task scheduling and handles resource management. The main objective of ROS for reconfigurable

platforms is to reduce the complexity of developing applications by giving the developer a higher level of abstraction with which to work.

Problem Definition. Performance is one of the fundamental reasons for using *Reconfigurable Computing Systems (RCS)*. By mapping algorithms and applications to hardware, designers can not only tailor the computation components, but also perform data-flow optimization to match the algorithm. Typically, a designer would manually divide the FPGA fabric into both *static* and *dynamic* regions [2]. The static regions accommodate modules that do not change in time, such as the task manager and necessary buses used for communication. The dynamic region is partitioned into a set of uniform or nonuniform blocks with a certain size called *Partial Reconfigurable Regions (PRRs)*. Each PRR can accommodate *Reconfigurable Modules (RM)* application specific hardware accelerators for the incoming tasks that need to be executed. In any type of operating system, a scheduler decides *when* to load new tasks to be executed. Efficient task scheduling algorithms have to take task dependencies, data communication, task resource utilization, and system parameters into account to fully exploit the performance of a dynamic reconfigurable system.

A scheduling algorithm for a reconfigurable system that assumes one architecture variant for the hardware implementation of each task may lead to inferior solutions [3]. Having multiple execution units per task helps in reducing the imbalance of the processing throughput of interacting tasks [4]. Also, static resource allocation for *Run-Time Reconfiguration (RTR)* might lead to inferior results, as the number of PRRs for one application might be different from the number required by another. The type of PRRs (uniform, nonuniform, and hybrid) tends to play a crucial role as well in determining both performance and power consumption (Figure 1).

ROS Main Components. Reconfigurable computing is capable of mapping hardware tasks onto a finite FPGA fabric while taking into account the dependency among tasks and timing constraints. Therefore, managing the resources becomes essential for any reconfigurable computing platform. Dynamic partial reconfiguration allows several independent bit-streams to be mapped into the FPGA fabric independently, as one architecture can be selectively swapped out of the chip while another is actively executing. Partial reconfiguration provides the capability of keeping certain parts intact in the FPGA, while other parts are replaced.

A Reconfigurable Computing System traditionally consists of a GPP and several Reconfigurable Modules (as seen in Figure 2) that execute dedicated hardware tasks in parallel [5]. A fundamental feature of a partially reconfigurable FPGA is that the logical components and the routing resources are time multiplexed. Therefore, in order for an application to be mapped onto an FPGA, it needs to be divided into independent tasks such that each subtask can be executed at a different time.

The use of an operating system for reconfigurable computing should ease application development, provide a higher level of abstraction, and, most importantly, verify and maintain applications. There are several essential modules that need to exist in any reconfigurable operating system implementation, including the bit-stream manager, scheduler, placer, and communications network (Figure 2).

(1) *The Scheduler*. This is an important component that decides when tasks will be executed. Efficient task scheduling algorithms have to take data communication, task dependencies, and resource utilization of tasks into consideration to optimally exploit the performance of a dynamic reconfigurable system. Scheduling techniques proposed in the literature have different goals, such as improving hardware resource utilization, reduction of scheduling time, and/or reconfiguration overhead. Other reconfigurable computing schedulers attempt to reduce fragmentation and power consumption [6].

(2) *Bit-Stream Manager*. This module manages the loading/unloading of partial bit-streams from storage media into a PRR. The bit-stream manager requires fast and fairly large storage media. Therefore, it is preferable to use a dedicated hardware module for this task. A bit-stream manager is further decomposed into a storage manager and a reconfigurable manager. The latter reads bit-streams from the storage manager and then downloads them onto the FPGA fabric.

(3) *The Placer*. In conventional reconfigurable devices, a placer decides where to assign new tasks in the reconfigurable area. In RTR systems, the scheduler and placer are interdependent. Therefore, they are implemented as a single entity that makes it challenging to identify clear boundaries between them.

(4) *Communication Manager*. This module defines how hardware and software tasks communicate and interact with each other. The communication network proposed by the manager depends on the type of applications used in the system. For example, streaming applications such as vision need a different topology than that of a centralized computational application.

Proposed Work and Motivations. In this paper, we propose and implement a framework for ROS that aids the designer from the early stages of the design to the actual implementation and submission of the application onto the reconfigurable fabric. A novel online scheduler, coined *RCOnline*, is developed that seeks to reuse hardware resources for tasks in order to reduce reconfiguration overhead, migrate tasks between software and hardware, and schedule operations based upon a priority scheme. With regard to the latter, RCOnline dynamically measures several system metrics, such as execution time and reconfiguration time, and then uses these metrics to assign a priority to each task. RCOnline then migrates and assigns tasks to the most suitable processing elements (SW or HW) based on these calculated priorities.

One of the main problems encountered in RTR is identifying the most appropriate floorplan or infrastructure that suits an application. Every application (e.g., machine vision,

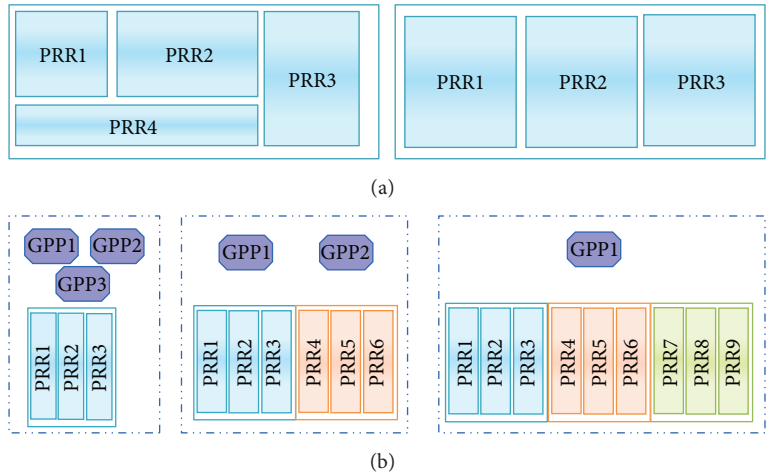


Figure 1: (a) PRR layout, (b) miscellaneous platforms/floorplans.

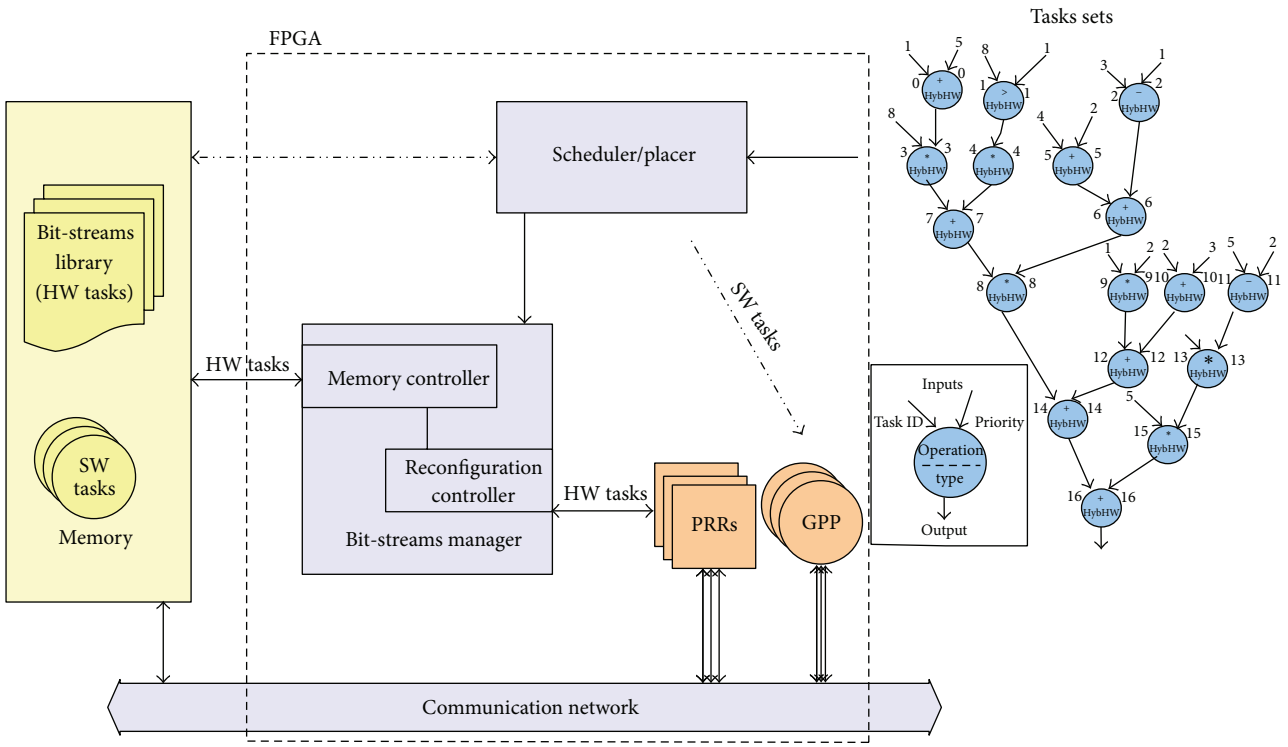


Figure 2: Reconfigurable OS: essential components.

wireless-sensor network) requires specific types of resources that optimize certain objectives, such as reducing power consumption and/or improving execution time. In our previous work published in [7], a dynamic reconfigurable framework consisting of five reconfigurable regions and two GPPs was implemented, along with simple scheduling algorithms. In this paper, a novel efficient and robust scheduling algorithm is developed that seeks to reuse hardware tasks to reduce reconfiguration time. The developed scheduler was tested using a collection of tasks represented by *Data-Flow Graphs* (DFGs). To verify the scheduler functionality and performance, DFGs with different sizes and parameters were required; therefore, a DFG generator was also designed and implemented. The DFG generator is able to generate benchmarks with different sizes and features. However, using such a dynamic reconfigurable platform to evaluate hundreds of DFGs based on different hardware configurations is both complex and tedious. The FPGA platform to be used in this work requires a different floorplan and bit-stream for each new configuration which limits the scope of testing and evaluation. Accordingly, an architecture reconfigurable simulator was also developed to simulate the hardware platform, while running the developed

reconfigurable operating system. The simulator allows for faster evaluation and flexibility and thus can support different hardware scenarios by varying the number of processing elements (PRRs, GPPs), size/shape of PRRs, and schedulers.

Contrary to software tasks, hardware tasks may have multiple implementation variants (execution units or hardware implementation). These variants differ in terms of performance, power consumption, and area. As limiting a scheduler to one execution variant may lead to inferior solutions, we also propose an efficient optimization framework that can mitigate the mentioned deficiency. The optimization framework extends the work we introduced in [8] which utilizes an evolutionary Island Based Genetic Algorithm technique. Given a particular DFG, each island seeks to optimize speed, power, and/or area for a different floorplan. The floorplans differ in terms of the number, size, and layout of PRRs, as described earlier. The framework presented in this paper uses the online proposed schedulers along with the Island Based GA Engine, for evaluating the quality of a particular schedule in terms of power and performance based on the most suitable floorplan for the intended application. This paper seeks to integrate the developed tools (simulator, DFG generator, and schedulers) along with the Genetic Algorithm Engine for task allocation in a unique framework that is capable of mapping and scheduling tasks efficiently on a reconfigurable computing platform.

Contributions. The main contributions of this paper can be summarized in the following points:

(1) *Operating System Support*. The development and evaluation of a novel heuristic for online scheduling of hard, real-time tasks for partially reconfigurable devices are achieved. The proposed scheduler uses fixed predefined Partial Reconfigurable Regions with reuse, relocation, and task migration capability. The scheduler dynamically measures several performance metrics such as reconfiguration time and execution time, calculates a priority for each task, and then assigns incoming tasks to the appropriate processing elements based on this priority. To evaluate the proposed framework and schedulers, a DFG generator and a reconfigurable simulator were also developed. The simulator is freely available under the open source GPL license on GitHub [9] to enable researchers working in the area of reconfigurable operating systems to utilize it as a supporting tool for their work.

(2) *Multiple Execution Variants*. A multiobjective optimization framework capable of optimizing total execution time and power consumption for static and partial dynamic reconfigurable systems is proposed. Not only is this framework capable of optimizing the aforementioned objectives, but also it can determine the most appropriate reconfigurable floorplan/platform for the application. This way, a good trade-off between performance and power consumption can be achieved which results in high energy efficiency. The proposed Island Based GA framework achieves on average 55.2% improvement over single-GA implementation and 80.7% improvement over a baseline random allocation and binding approach. The GA framework is made freely available under the open source GPL license on GitHub [10]. To the best of our knowledge, this is the first attempt to use multiple GA instances for optimizing several objectives and aggregating the results to further improve solution quality.

Paper Organization. The remainder of this paper is organized as follows. Section 2 discusses the most significant work published in the field of scheduling and execution unit assignments for reconfigurable systems. Section 3 introduces the tools developed to test and evaluate the proposed work. Section 4 introduces a novel reconfigurable scheduler with a comparison with other schedulers adapted from the literature. Section 5 introduces the evolutionary framework developed for allocating and binding execution units to task graphs. Finally, conclusions and future directions are presented in Section 6.

2. Related Work

The work in [11, 12] first proposed the idea of using reconfigurable hardware, not as a dependent coprocessor, but as a standalone independent hardware unit. In this early implementation hardware tasks can access a FIFO, memory block, and I/O driver or even signal the scheduler. The system uses an external powerful processor and connects to an FPGA via the PCI port. The proposed run-time environment can dynamically load and run variable size tasks utilizing partial reconfiguration. The system proposed in [12] was used to explore online scheduling for real-time tasks for both 1D and 2D modules.

The work in [13] was the first to explore the migration between hardware and software tasks. The authors studied the coexistence of hardware (on an FPGA) and software threads and proposed a migration protocol between them for real-time processing. The authors introduced a novel allocation scheme and proposed an architecture that can benefit from utilizing dynamic partial reconfiguration. However, the results published were mainly based on pure simulation.

In [14], the author presented a run-time hardware/software scheduler that accepts tasks modeled as a DFG. The scheduler is based on a special architecture that consists of two processors (master and slave), a Reconfigurable Computing Unit (RCU), and shared memory. The RCU runs hardware tasks and can be partially reconfigured. Each task can be represented in three forms: a hardware bit-stream run on the RCU, a software task for the master processor, or a software task for the slave processor. A scheduler migrates tasks between hardware and software to achieve the highest possible efficiency. The authors found the software implementation of the scheduler to be slow. Therefore, the authors designed and implemented a hardware version of the scheduler to minimize overhead. The work focuses mainly on the hardware architecture of the scheduler and does not address other issues associated with

the loading/unloading of hardware tasks or the efficiency of intertask communication. Another drawback was the requirement that each task in the DFG be assigned to one of the three processing elements mentioned above. This restriction conflicts with the idea of software/hardware migration.

The authors of [15] designed and implemented an execution manager that loads/unloads tasks in an FPGA using dynamic partial reconfiguration. The execution manager insures task dependency and proper execution. It also uses prefetching and task reuse to reduce configuration time overhead. The task is modeled using scheduled DFGs in order to be usable by the execution manager. The authors implemented a software and hardware version of the manager but found the software versions to be too slow for real-time applications. The implementation used a simulated hardware task that consists of two timers to represent execution and configuration time. There are still many unaddressed issues for the manager to be used with real tasks, such as configuration management and intertask communication.

The work in [16] proposes a hardware OS, called CAP-OS, that uses run-time scheduling, task mapping, and resource management on a run-time reconfigurable multiprocessor system. The authors in [17] proposed to use the RAMPSOC with two schedulers: a static scheduler to generate a task graph and assign resources and a dynamic scheduler that uses the generated task graph produced by the static scheduler. A deadline constraint for the entire graph is imposed instead of individual tasks. Most of their work was based on simulation, and the implementation lacked task reconfiguration.

Many researchers considered the use of multiple architecture variants as part of developing an operating system or a scheduler. In [18, 19] multiple core variants were swapped to enhance execution time and/or reduce power consumption. In our previous work [7], we alternated between software and hardware implementation of tasks to reduce total execution time. The work presented here is different in its applicability since it is more comprehensive. It can be used during the design stage to aid the designer to select the appropriate application variant for tasks in the task graph, as will be described in Section 5.

Genetic Algorithms (GAs) were used by many researchers to map tasks into FPGAs. For example, in [20] a hardware based GA partitioning and scheduling technique for dynamically reconfigurable embedded systems was developed. The authors used a modified list scheduling and placement algorithm as part of the GA approach to determine the best partitioning. The work in [21] maps task graphs into a single FPGA composed of multiple Xilinx MicroBlaze processors and hardware accelerators. In [22], the authors used a GA to map task graphs into partially reconfigurable FPGAs. The authors modeled the reconfigurable area into tiles of resources with multiple configuration controllers operating in parallel. However, all of the above works do not take into account multiple architecture variants.

In [18] the authors modified the *Rate Monotonic Scheduling (RMS)* algorithm to support *hot swapping* architecture implementation. Their goal is to minimize power consumption and reschedule the task on the fly while the system is running. The authors addressed the problem from the perspective of schedule feasibility, but their approach does not target any particular reconfigurable platform, nor does it take multiple objectives into consideration which makes it different from the proposed work in this paper. In [23], the authors used an *Evolutionary Algorithm (EA)* to optimize both power and temperature for heterogeneous FPGAs. The work in [23] is very different from the current proposed framework in terms of the types of objectives considered and their actual implementation, since the focus is on swapping a limited number of GPP cores in an embedded processor for the arriving tasks. In contrast, our proposed work focuses on optimizing the hardware implementation for every single task in the incoming DFG and targets static and partial reconfigurable FPGAs.

In [3, 4] the authors used a GA framework for task graph partitioning along with a library of hardware task implementation events that contain multiple architecture variants for each hardware task. The variants reflect trade-offs between hardware resources and task execution throughput. In their work, the selection of implementation variants dictates how the task graph should be partitioned. There are a few important differences from our proposed framework. First, our work targets dynamic partial RTR with reconfigurable operating systems. The system simultaneously runs multiple hardware tasks on PRRs and can swap tasks in real-time between software and hardware (the reconfigurable OS platform is discussed in [7]). Second, the authors use a single objective approach, selecting the variants that give minimum execution time. In contrast, our approach is multiobjective and not only seeks to optimize speed and power, but also seeks to select the most suitable reconfigurable platform. As each of the previous objectives is likely to have its own optimal solution, the resulting optimization does not give rise to a single superior solution, but rather a set of optimal solutions, known as Pareto optimal solutions [24]. Unlike the work in [3, 4], we employ a parallel, multiobjective Island Based GA to generate the set of Pareto optimal solutions.

3. Methodology

In this section, the overall methodology of the proposed work, along with the tools developed to implement the framework, is presented. The two major phases of the flow are shown in Figure 3:

(1) *Scheduling and Placement*. This phase is responsible for scheduling and placing incoming tasks on the available processing elements (GPP/PRRs). In our earlier work in [7], a complete reconfigurable platform, based on five PRRs and two GPPs which runs a simple ROS, was designed and mapped onto a Xilinx 6 FPGA. For our current work, using the previously developed hardware reconfigurable platform imposed a limitation, as modifying the PRRs/GPPs is a tedious and time-consuming process. Accordingly, we developed a reconfigurable simulator that hosts the developed ROS and emulates the reconfigurable platform, as will be discussed in Section 3.2. The ROS

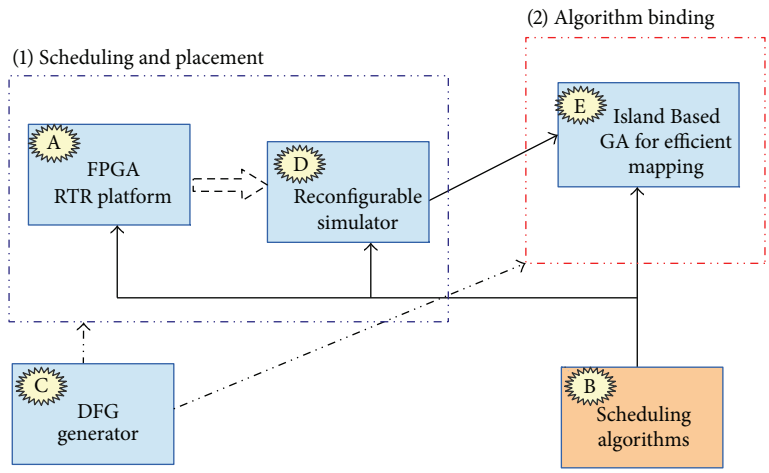


Figure 3: Overall methodology.

runs a novel online reconfigurable-aware scheduler (RCOnline) that is able to migrate tasks between software and hardware and measure real system metrics to optimize task scheduling, as will be explained in Section 4.2. We also developed a DFG generator to evaluate the proposed system, which will be presented in Section 3.1.

(2) *Algorithm Binding*. Phase one assumes a single execution element (architecture) per task. This assumption might lead to a suboptimal solution, as a hardware task can be implemented in different forms (e.g., serial or parallel implementation). To allow the proposed system to consider multiple (different) architecture implementation events, we implemented a multiobjective Island Based Genetic Algorithm platform that not only optimizes for power and speed, but also selects the most appropriate platform for the DFG, as will be explained in Section 5.

3.1. DFG Generator. Scheduling sequential tasks on multiple processing elements is an NP-hard problem [25]. To develop, modify, and evaluate a scheduler, extensive testing is required. Scheduling validation can be achieved using workloads created either by actual recorded data from user logs or more commonly by use of randomly synthetically generated benchmarks [25]. In this work, a synthetic random DFG generator was developed to assist in verifying system functionality and performance (note: several real-world benchmarks from MediaBench DSP suite [26] were also used to validate the work presented, as will be explained in Section 3.3). An overview of the DFG generator is presented in Figure 4. The inputs to the DFG generator include a library of task types (operations), the number of required nodes, the total number of dependencies between nodes (tasks), and the maximum number of dependencies per task. In addition, the user can control the percentage of occurrences for each task or group of tasks. The outputs of the DFG generator consist of two files: a graph and a platform file. The graph file is a graphical representation of the DFG in a user-friendly format [27], while the platform file is the same DFG in a format compatible with the targeted platform.

The DFG generator consists of three major modules, as seen in Figure 4:

(1) *Matrix Generator*. This module provides the core functionality of the DFG generator and is responsible for constructing the DFG skeleton (i.e., the nodes and dependencies). The nodes and edges that constitute the DFG are generated randomly. The nodes represent functions (or tasks) and the edges in the graph represent data that is communicated among tasks. Both tasks and edges are generated randomly based on user-supplied directives. The matrix dependency generator builds a DFG without assigning *operations* to the nodes (nodes and dependencies).

(2) *Matrix Operation Generator*. This module randomly assigns task types to the DFG (nodes) generated by the *Matrix Generator* module from a library of task types based on user-supplied parameters.

(3) *Graph File Generator*. This module produces a graph representation of the DFG in a standard open source *DOT* (graph description language) format [27].

3.2. Reconfigurable Simulator. Using the hardware platform introduced in [7] has several advantages but also introduces some limitations. The FPGA platform to be used in this work requires a different floorplan and bit-stream for each new configuration which limits the scope of testing and evaluation. Accordingly, an architecture reconfigurable simulator was developed to simulate the hardware platform in [7], while running the developed reconfigurable operating system. The simulator consists of three distinct layers, as shown in Figure 5.

The *Platform Emulator Layer (PEL)* emulates the RTR platform functionality and acts as a virtual machine for the upper layers. The ROS kernel along with the developed schedulers runs on top of the PEL. The developed code of the ROS kernel along with the schedulers can run on both

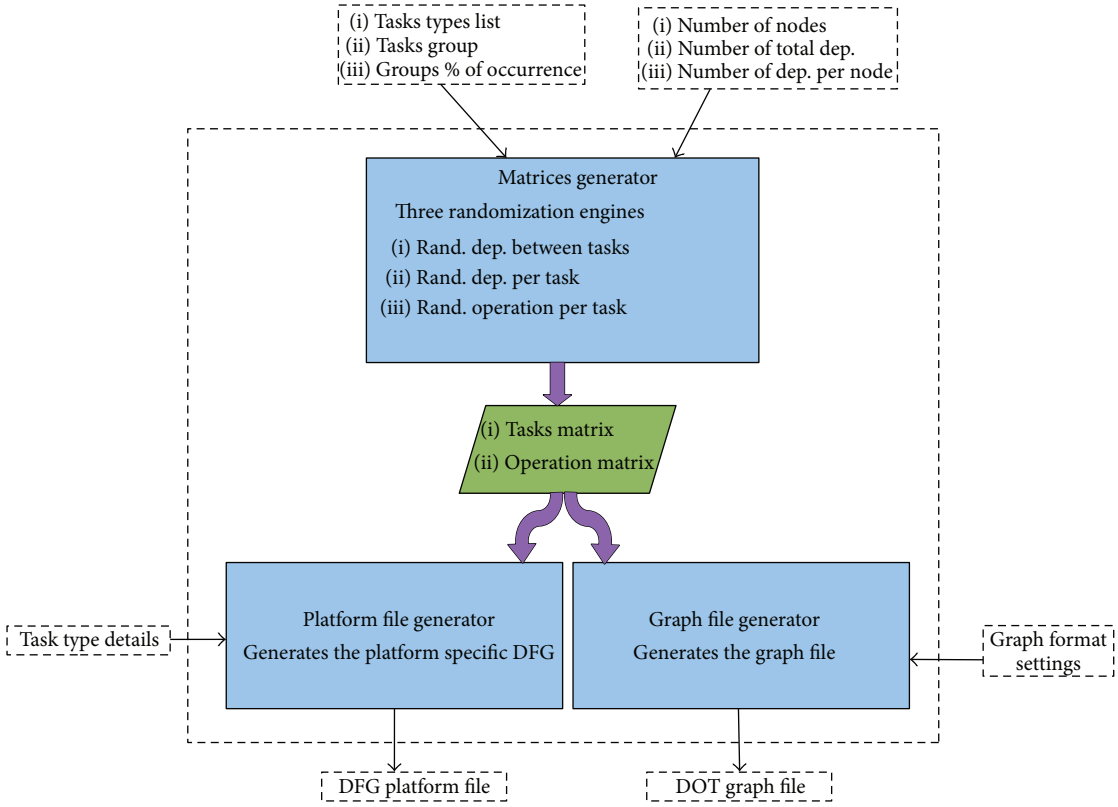


Figure 4: Components of the DFG generator.

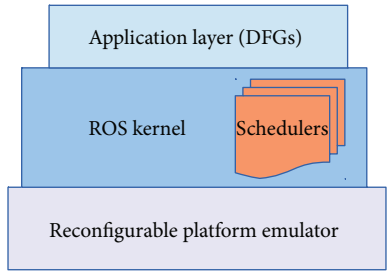


Figure 5: Reconfigurable simulator layout.

the simulator and the actual hardware RTR platform without any modification. The use of the PEL layer enables the designer to easily develop and test new schedulers on the simulator before porting them to the actual hardware platform.

The developed simulator utilizes the proposed schedulers and supports any number of PRRs and/or GPPs (software or hardware). The simulator was written in the C language and runs under Linux. The code was carefully developed in a modular format, which enables it to accommodate new schedulers in the future. The simulator uses several configuration files, similar to the one used by the Linux OS. We opted to make the simulator freely available under the open source GPL license on GitHub [9] to enable researchers working in the area of reconfigurable operating systems to utilize it as a supporting tool for their work.

3.2.1. Simulator Inputs. The simulator accepts different input parameters to control its operation including scheduler selection and architecture files names. The simulator is developed to emulate the hardware platform and expects the following configuration files as input:

(i) A *task (architecture) library* file which stores task information used by the simulator: task information includes the mode of operation (software, hardware, or hybrid), execution time, area, reconfiguration time, reconfiguration power, and dynamic power consumption (hybrid tasks can migrate between hardware and software), on a per-task-type basis. Some of these values are based on analytic models found in [28, 29], while others are measured manually from actual implementation on a Xilinx Virtex-6 platform.

(ii) A *layout (platform)* file which specifies the FPGA floorplan: the layout includes data that represent the size, shape, and number of PRRs along with the types and number of GPPs.

(iii) A *DFG* file which stores the DFGs to be scheduled and executed.

3.2.2. Simulator Output. The simulator generates several useful system parameters, including total execution time,

Table 1: Benchmark specifications.

ID	Name	# of nodes	# of edges	Avg. edges/node	Critical path	Parallelism (nodes/crit. path)
S1	Synthesized (4 task types)	50	50	1	6	8.3
S2	Synthesized (4 task types)	150	200	1.33	15	10.1
S3	Synthesized (8 task types)	25	40	1.6	7	3.5
S4	Synthesized (8 task types)	100	120	1.2	7	14.2
S5	Synthesized (5 task types)	13	10	0.77	3	4.3
S6	Synthesized (4 task types)	10	5	0.5	3	3.3
S7	Synthesized (8 task types)	50	60	1.2	8	6.3
S8	Synthesized (4 task types)	100	120	1.2	7	14.3
S9	Synthesized (8 task types)	150	200	1.33	8	25.1
S10	Synthesized (8 task types)	200	60	0.3	4	50.1
S11	Synthesized (4 task types)	100	150	1.5	8	12.5
DFG2	JPEG-Smooth Downsample	51	52	1.02	16	3.2
DFG6	MPEG-Motion Vectors	32	29	0.91	6	5.3
DFG7	EPIC-Collapse_pyr	56	73	1.3	7	8.1
DFG12	MESA-Matrix Multiplication	109	116	1.06	9	12.1
DFG14	HAL	11	8	0.72	4	2.8
DFG16	Finite Input Response Filter 2	40	39	0.975	11	3.6
DFG19	Cosine 1	66	76	1.15	8	8.3

reconfiguration time (cycles), task migration information, and hardware reuse. A task placement graph can also be generated by enabling specific options as will be seen in the example presented in Section 5.2.2.

3.3. Benchmarks. Verifying the functionality of any developed scheduler with the aid of a simulator requires testing based on benchmarks with different parameters, properties, and statistics. Accordingly, ten DFGs were generated by the DFG generator presented in Section 3.1. These DFGs differ with respect to the number of tasks (nodes), dependencies, and operations, as shown in Table 1. The number of operations refers to the different unique functions (task types) in the DFG. The benchmark suite consists of two different sets of DFGs. The first set uses four operations, while the second set uses eight operations that have variable execution time, dependencies, and possible hardware/software implementation. In addition to the synthesized benchmarks, seven real-world benchmarks selected from the well-known MediaBench DSP suite [26] were used in the schedulers' evaluation, as shown in the lower part of Table 1.

4. Scheduling Algorithms

In this section, we introduce a novel online scheduling algorithm (RCOnline) along with an enhanced version (RCOnline-Enh.) for reconfigurable computing. The proposed scheduler manages both hardware and software tasks. We also introduce an efficient offline scheduler (RCOffline) for reconfigurable systems that is used to verify the quality of solutions obtained by the online scheduler. Offline schedulers often produce solutions superior to those obtained by online schedulers since the former build complete information about the system activities. In addition, several works in the related literature resort to such an approach to measure the quality of solutions obtained by online schedulers due to lack of open source code of work published in the past [30–32]. Also, schedulers developed by many researchers are usually implemented based on different assumptions and constraints that are not compatible with those employed in our proposed schedulers.

4.1. Baseline Schedulers. One of the main challenges faced in this work is the lack of compatible schedulers, tools, and benchmarks that can be used to compare with our results. Therefore, throughout the research process, we considered several published algorithms and adapted them to represent our RTR platform for the sole sake of comparing with our proposed schedulers. Two of the adapted scheduling algorithms are described briefly in the following sections.

4.1.1. RCOffline Scheduler. An offline scheduler was implemented and integrated into our proposed RTR platform. The RCOffline scheduler is a greedy offline reconfiguration aware scheduler for 2D dynamically reconfigurable architectures. It supports task reuse, module prefetch, and a defragmentation technique. Task reuse refers to a policy that the operating system can take to reduce reconfiguration time. This is accomplished by reusing the same hardware accelerator that exists in the floorplan of the FPGA when a new arriving task needs to be scheduled. The main advantage of utilizing such a scheduler is that it produces solutions (schedules) that are of high quality (close to those obtained by an ILP based model [33, 34]) in fraction of the time. The RCOffline

```
(1)  DFGs = Compute ALAP and sort (DFG).
(2)  ReadyQ =findAvailableTasks(DFGs)//no dependencies
(3)  while there are unscheduled tasks do
(4)    if (Reuse = true)
(5)          for all tasks in ReadyQ do
(6)                PRR# = FindAvailablePRR(task)
(7)                if (PRR#) then
(8)                      ts=t
(9)                      for all p in PredecessorOf(task)
(10)                           ts=max(ts, tps + tpl)
(11)                     end for
(12)                     remove task from ReadyQ
(13)               end if
(14)         end for
(15)   else /*Reuse = false*/
(16)         for all tasks in ReadyQ do
(17)               PRR# = findEmptyPRR(task)
(18)               if (PRR# and Reconfiguration) then
(19)                     ts=t+trs
(20)                     for all p in PredecessorOf(task)
(21)                           ts=max(ts, tps + tpl)
(22)                     end for
(23)                     Remove task from ReadyQ
(24)               end if
(25)         end for
(26)   end if
(27)   ReadyQ=CheckforAvailableTasks /* dependencies met */
(28) end while
```

Algorithm 1: Pseudo-code for RCOffline.

pseudo-code is shown in Algorithm 1. The RCOffline starts by sorting the incoming tasks in the DFG using the *As-Late-As-Possible (ALAP)* algorithm and then adds the ready tasks (no dependencies) to the ready queue (ReadyQ) [lines (1)-(2)]. For task reuse, the scheduler will locate an available PRR [line (6)] and then tentatively set the start time (t_s) to the current time (t) [line (8)]. Similarly, if task reuse is false, the scheduler will look for an empty or unused PRR and only perform reconfiguration if the reconfiguration controller is available [lines (17)-(18)]. The tentative start time in this case is set to the sum of current time and reconfiguration time (t_{rs}). To prevent running a task before the predecessors assigned to the same PRR, the scheduler checks the end time of the predecessor tasks [lines (9)–(11), (20)–(22)]. This might lead to a gap between the reconfiguration time of a task and the time to execute it on the PRR (i.e., task prefetching). Since RCOffline is an offline based scheduler it expects the entire DFG in advance. Also, it assumes the existence of a reconfigurable platform that supports task relocation and PRR resizing at run-time.

4.2. RCOnline Scheduler. The quality of the employed online scheduling algorithms in an RTR-OS can have a significant impact on the final performance of applications running on a reconfigurable computing platform. The overall time taken to run applications on the same platform can vary considerably by using different schedulers.

The online scheduler (RCOnline) proposed in this work was designed to intelligently identify and choose the most suitable processing element (PRR or GPP) for the current task based on knowledge extracted from previous tasks' historical data. The scheduler dynamically calculates task *placement priorities* based on data obtained from the previous running tasks. During task execution, the scheduler attempts to learn about running tasks by classifying them according to their designated types. The scheduler then dynamically updates a *"task type table"* with the new task type information. The data stored in the table is used by the scheduler at a later time to classify and assign priorities to new incoming tasks. The RCOnline scheduler was further enhanced by modifying task selection from the *ready queue*. This enhancement dramatically improves performance, as will be explained later in Section 4.4.

Enhancing algorithm performance by learning from historical previous data is not an issue for embedded systems since usually the same task sets run repeatedly on the same platform. For example, video processing applications use the same repeated task sets for every frame. That is to say, the scheduler will always produce a valid schedule even during the learning phase; however its efficiency (schedule quality) enhances over time.

The following definitions are essential to further understand the functionality of the RCOnline algorithm:

```
(1)  // The PRRS should be stored from the smallest to largest such that
(2)  // those with smaller reconfiguration time have smaller IDs
(3)  Sort PRRs in ascending order based on size.
(4)  Read the node with the highest task priority
(5)  if (All dependencies are met)
(6)          {add task to ready queue}
(7)  // SELECTION BY RCOnline versus RCOnline-Enh
(8)  Fetch the first task from the ReadyQueue // -- used by RCOnline
(9)  Fetch a task that matches an existing reconfigured task // -- used by RCOnline-Enh
(10) switch (task->mode)
(11) {  case Hybrid_HW :
(12)    {   if(TaskTypePriority==0 and GPP is free)
(13)              {// TaskTypePriority ==0 indicates that task is preferred to run on SW
(14)           Migrate task from HybridHW to HybridSW
(15)               Add to the beginning of the ReadyQueue}
(16)      }else{
(17)      if (there is a free PRR(s) that fits the current task)
(18)       {{Search Free PRRs for a match
(19)           against the ready task}
(20)      if (task found)
(21)              run ready task on the free PRR
(22)      else{
(23)          currentPRR= smallest free PRR;
(24)           if(TaskTypePriority< currentPRR and GPP is free)
(25)            {  // check if it's faster to reconfigure or run on SW
(26)          Migrate task from HybridHW to HybridSW
(27)            Add to the beginning of the ReadyQueue }
(28)        }else{
(29)            Reconfigure currentPRR with the ready task bit-stream}}
(30)   }else // all PRRs are busy
(31)     {if ( GPP is free)
(32)      {  Migrate task from HybridHW to HybridSW
(33)         Add to the beginning of the ReadyQueue }
(34)      }else{
(35)        // return to main program and
(36)           // wait for free resources
(37)           Increase the Busy Counter
(38)        return Busy}
(39)    break;  }
(40)   case HybridSW :
(41)    {if (GPP is free)
(42)      { load task into GPP}
(43)     }else if (there is a free PRR)
(44)      {  Migrate task from HybridSW to HybridHW
(45)         Add to the beginning of the ReadyQueue}
(46)     }else{
(47)       // return to main program and
(48)       //wait for free resources
(49)       Increase the Busy Counter
(50)       return Busy
(51)       break;}} End
```

Algorithm 2: Pseudo-code for RCOnline/RCOnline-Enh. (with reuse and task migration).

(i) *PRRs Priority*. It is calculated based on the PRR size. PRRs with smaller size have lower reconfiguration time and thus have higher priorities.

(ii) *Placement Priority.* It is a dynamically calculated metric which is frequently updated on the arrival of new tasks. Placement priority is assigned based on the type (function) of task. RCOnline uses this metric to decide *where* to place the current tasks. Placement priority is an integer value, where the lower the number the higher the priority.

Based on the pseudo-code of Algorithm 2, RCOnline takes the following steps to efficiently schedule tasks:

(1) The PRRs are given priority based on their respective size. PRR priority is a number from 0 to $\text{PRR}_{\text{max}} - 1$.

(2) Arriving tasks are checked for dependencies, and if all dependencies are satisfied, the task is appended to the *ready queue*.

(3) The scheduler then selects the task with the highest priority from the ready queue; if *task mode* is *HybridHW* (i.e., it can be placed in either HW or SW) then the following occur:

(a) The scheduler checks the current available PRRs for task reuse. If reuse is not possible (i.e., no module with similar architecture is configured), the scheduler uses *placement priority* and *PRR priority* to either place the task on an alternative PRR or migrate it to software (lines (28) to (41) in the pseudo-code of Algorithm 2). This step is useful for nonuniform implementation, since each PRR has a different reconfiguration time. Task migration is performed when the following conditions are met:

```
Migrate_task= True,
if [(HW Exec. time + Reconfig.
time) > SW Exec. Time ] or
[there are No free PRRs]
```

(b) If no resources are available (either in hardware or in software), a busy counter is incremented while waiting for a resource to be free (lines (46) to (50)). The *busy counter* is a performance metric that is used to give an indication of how busy (or idle) the scheduler is. The busy counter counts the number of cycles a task has to wait for free resources.

(4) When the task mode is HybridSW (i.e., task should run on SW but can also run on HW), RCOnline first attempts to run the task on a GPP and then migrates it to hardware based on GPP availability. Otherwise, RCOnline increments the busy counter and waits for free resources (lines (49) to (50)).

(5) Software and hardware only tasks are handled in a similar way to HybridSW and HybridHW, respectively, with no task migration.

Placement priority, which is calculated by the scheduler, is different than *task priority*, which is assigned to DFG nodes at design time, in that placement priority determines where a ready task should be placed, while *task priority* determines the order in which tasks should run.

The calculation of the PRR placement priority takes into account the PRRs' reconfiguration times in addition to task execution time:

(1) At the end of task execution, RCOnline adds any newly discovered *task type* to the *task type table*.

(2) The scheduler updates the reconfiguration time for the current PRR in the *task type table*.

(3) RCOnline updates the execution time for the current processing element based on the following formula:

$$E_{\text{new}} = E_{\text{old}} + \left(E_{\text{new}} - E_{\text{old}}\right) * X, \tag{1}$$

where E denotes execution time and X is the learning factor. Based on empirical testing and extensive experimentation, X has been set to a value of 0.2. The basic idea is to adapt the new execution time in such a way that does not radically deviate away from the average execution times of the processing element. This assists the scheduler to adapt for changes in execution time for the same operation. The learning formula takes into account old accumulated average execution time (80%) and the most recent instance of the task (20%). The goal of the *task type table* is to keep track of execution times (and reconfiguration times in the case of PRRs), of each task type on every processing element (PRRs and GPPs). Accordingly, when similar new tasks arrive, the scheduler can use the data in the table to determine the appropriate location to run the task.

(4) Finally, RCOnline uses the measured reconfiguration and execution times to calculate placement priority for each task type.

Assuming all tasks in a DFG are available, the complexity of RCOnline is $O(n)$, where n is the number of tasks in the DFG. Figure 6 depicts the learning behavior of RCOnline with an example DFG consisting of six nodes. As *task 1* completes execution, the system records the following information (as shown in Figure 6(a)) in the *task type table*: (i) the task operation (multiplication in this case), (ii) reconfiguration time (10 ms in PRR1), and (iii) the execution time (HW, 200 ms). Following the execution of *task 2*, *task type table* is updated again with the features of the new task type (addition), as shown in Figure 6(b). As tasks 4, 5, and 6 arrive, the scheduler will have had more knowledge about their reconfiguration and execution times based on prior information collected. The recorded performance metrics are then used to calculate *placement priority* that will assist the scheduler to efficiently select future incoming tasks with similar type.

4.3. RCOnline-Enh. To further enhance the performance of the RCOnline scheduler, the task selection criteria were slightly modified. Instead of selecting the *first* available task in the ready queue (i.e., the task with the highest priority), the scheduler searches the ready queue for a task that matches an existing reconfigured task (line (9) in Algorithm 2). In practice, this simple modification dramatically reduces the number of reconfigurations (see the results presented in Section 4.4). Figure 7 clearly demonstrates the main difference between the RCOnline algorithm and the RCOnline-Enh. As can be seen in the example, the RCOnline-Enh. seeks to switch task "2" with task "1" to take advantage of task reuse and thus minimize the overall reconfiguration time that takes place in the system.

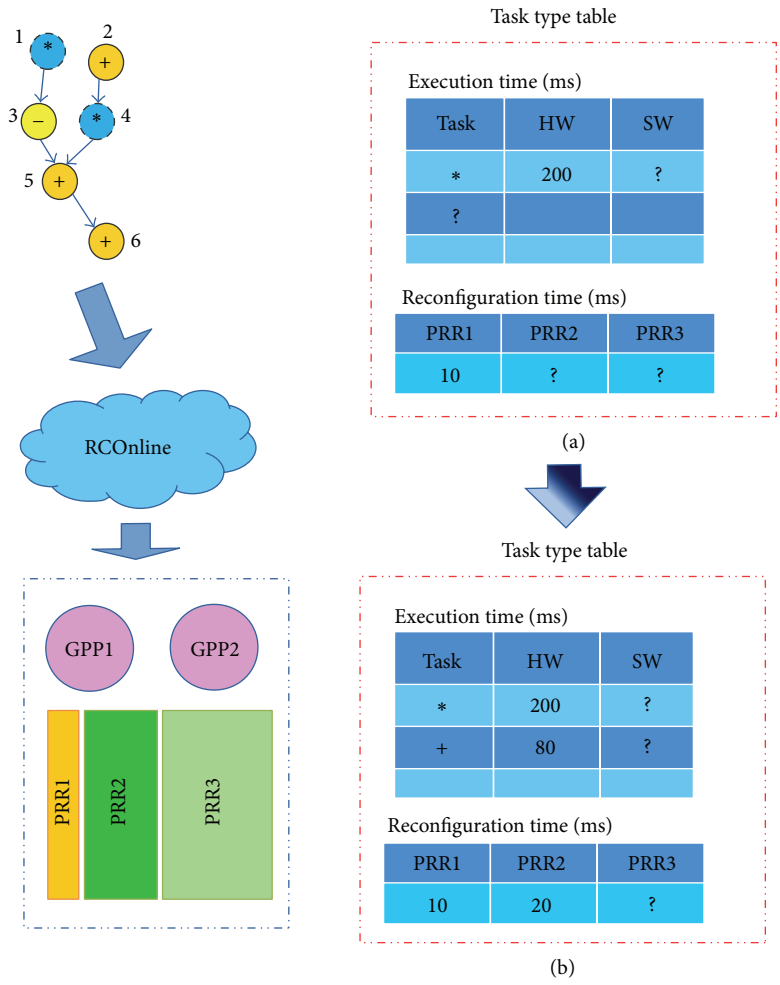


Figure 6: Learning in RCOnline scheduler.

4.4. Results and Analysis. In this section, results based on RCOnline along with an enhanced version of RCOnline are presented. In order to achieve a fair comparison with the RCOffline scheduler, task migration and software tasks were completely disabled from RCOnline and RCOnline-Enh. implementation. The same reconfigurable area available in the hardware platform is assigned to the RCOffline scheduler.

4.4.1. Total Execution Time. The total execution time or schedule time (cycles) measured for the proposed online schedulers (RCOnline) and its variant are presented in Tables 2 and 3, respectively. The tables compare the solution quality obtained by RCOnline to the solutions obtained by RCOffline, as well as to the solutions found by RCOnline-Enh. C_{off}, C_{on}, and C_{enh} are the total execution times (in clock cycles) found by RCOffline, RCOnline, and RCOnline-Enh., respectively. $\Delta C = 100(C_{off} - C_{on})/C_{on}$ and $\Delta C_{enh} = 100(C_{off} - C_{enh})/C_{enh}$ is the relative error in percent for the schedule length found by RCOnline and RCOnline-Enh. compared to RCOffline. This notation is used in all tables presented in this section. Table 2 shows the *total execution time* of the first run for all schedulers. Results are based on the assumption that all PRRs are empty and RCOnline is in the initial phase of learning. Table 3, on the other hand, shows the *total execution time* after the first two iterations. At this point, the learning phase of RCOnline is complete (i.e., third iteration). Since RCOffline is an offline scheduler with no learning capability, the time it takes to configure the first task of a DFG was subtracted (assuming it is already configured).

The results in Table 2 clearly indicate an average performance gap increases by 40% when comparing RCOnline-Enh. to RCOnline. The performance gap drops to −6% compared to RCOffline. This is a significant improvement, taking into account the fact that RCOnline-Enh. is an online scheduler that requires a small CPU time compared to the RCOffline. The RCOnline-Enh. performance is improved dramatically following the learning phase, as shown in Table 3. The RCOnline-Enh. performance gap was 49% of RCOnline and 2% of RCOffline. The performance is expected to improve even more if task migration is enabled.

The CPU run-time comparison of RCOnline-Enh. and RCOffline is shown in Table 4. The tests were performed on a Red Hat Linux workstation with 6-core Intel Xeon processor, running at a speed of 3.2 GHz and equipped with 16 GB of

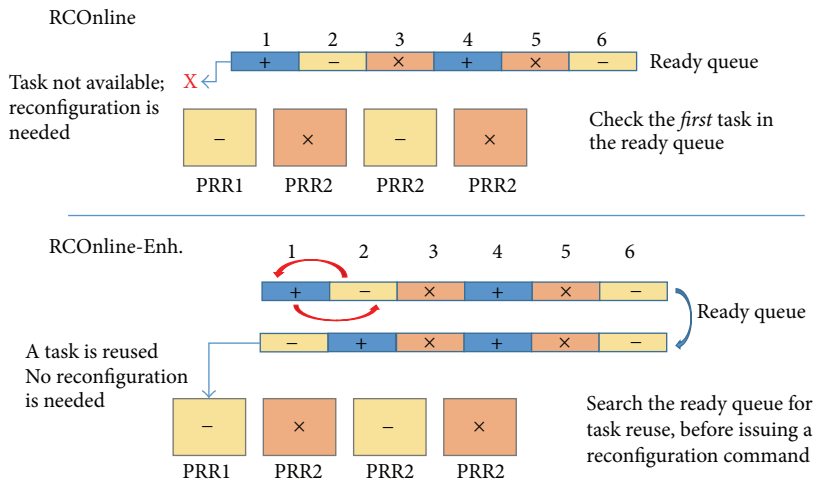


Figure 7: RCOnline versus RCOnline-Enh.

Table 2: Schedulers comparison for *total execution time* (no learning).

Benchmark	Total execution time			Percentage	
	C_{off}	C_{on}	C_{enh}	ΔC	ΔC_{enh}
S1	520	740	540	−30	−4
S2	1160	1750	1220	−34	−5
S3	300	420	340	−29	−12
S4	775	1470	820	−47	−5
S6	160	160	160	0	0
S7	420	860	500	−51	−16
S8	380	500	460	−24	−17
S9	995	1990	1000	−50	−1
S10	1095	3000	1140	−64	−4
S11	920	1430	1000	−36	−8
DFG2	680	800	584	−15	16
DFG6	325	340	359	−4	−9
DFG7	405	600	481	−33	−16
DFG12	745	910	761	−18	−2
DFG14	140	160	142	−13	−1
DFG16	295	403	383	−27	−23
DFG19	455	540	460	−16	−1
Average	**575**	**945**	**609**	**−29**	**−6**

Table 3: Schedulers comparison for *total execution time* (after learning phase).

Benchmark	Total execution time			Percentage	
	C_{off}	C_{on}	C_{enh}	ΔC	ΔC_{enh}
S1	500	660	520	−24	−4
S2	1140	1690	1160	−33	−2
S3	280	390	280	−28	0
S4	755	1370	660	−45	14
S6	140	140	120	0	17
S7	400	700	430	−43	−7
S8	360	440	345	−18	4
S9	975	1950	950	−50	3
S10	1075	2620	1075	−59	0
S11	900	1230	866	−27	4
DFG2	660	830	584	−20	13
DFG6	305	360	322	−15	−5
DFG7	385	555	436	−31	−12
DFG12	725	870	723	−17	0
DFG14	120	160	103	−25	17
DFG16	275	380	288	−28	−5
DFG19	435	540	460	−19	−5
Average	**555**	**876**	**548**	**−28**	**2**

RAM. The RCOnline-Enh. is on average 10 times faster than the RCOffline.

4.4.2. Hardware Task Reuse. Tables 5 and 6 show the number of reused hardware tasks prior to the learning phase and after learning phase, respectively. Hardware task reuse tends to reduce the number of required reconfigurations and, hence, total execution time. It is clear from the tables that the amount of hardware task reuse was improved with RCOnline-Enh. Obviously, this contributed to a reduction in the total execution time. Table 6 shows the benefit of the learning phase which tends to increase the variety of the available reconfigured tasks.

The RCOnline-Enh. has on average 172% and 158% more hardware reuse than RCOnline prior to and following the learning phase, respectively. RCOnline-Enh. exceeded the amount of hardware reuse over RCOffline after the learning phase by 104%, and it reached 87% of the hardware reuse without learning. RCOffline is able to prefetch tasks, which influences the amount of reuse. All results presented in this section were based on scheduling tasks within a DFG using a single type of architecture per task and based on a fixed single floorplan. The next section shows how these

Table 4: Run-time comparison of RCOffline and RCOnline-Enh.

Benchmark	Time in (ms)		Speedup (X times)
	RCOffline	RCOnline-Enh.	
S1	12	1.52	7.9
S2	59	9.14	6.4
S3	5	0.418	12
S4	29	2.03	14.3
S6	2	0.116	17.0
S7	8	1.31	6.0
S8	5	0.614	8.0
S9	58	6.28	9.2
S10	110	9.74	11.3
S11	31	4.98	6.2
DFG2	12	0.829	14.5
DFG6	3	0.553	5.4
DFG7	7	1.15	6.1
DFG12	29	3.69	7.9
DFG14	2	0.111	18.0
DFG16	3	0.754	3.9
DFG19	12	1.39	8.6
Average	**23**	**3**	**9.5**

Table 5: Comparison of schedulers based on number of reused tasks (before learning).

Benchmark	Number HW task reuse			Percentage	
	C_{off}	C_{on}	C_{enh}	ΔC	ΔC_{enh}
S1	38	22	37	−42	−3
S2	139	74	128	−47	−8
S3	17	6	10	−65	−41
S4	74	28	67	−62	−9
S6	5	4	4	−20	−20
S7	36	8	27	−78	−25
S8	18	9	14	−50	−22
S9	128	53	118	−59	−8
S10	181	54	173	−70	−4
S11	88	40	78	−55	−11
DFG2	37	23	38	−38	3
DFG6	21	19	19	−10	−10
DFG7	49	36	42	−27	−14
DFG12	86	72	84	−16	−2
DFG14	7	4	5	−43	−29
DFG16	32	26	26	−19	−19
DFG19	50	42	49	−16	−2
Average	**59**	**31**	**54**	**−42**	**−13**

schedulers can be used as part of a multiobjective framework to determine the most appropriate architecture from a set of variant architectures for a given task. The framework proposed in the next section not only is capable of optimizing the selection of the architecture for a task within a DFG,

Table 6: Comparison of schedulers based on number of reused tasks (after learning).

Benchmark	Number or task reuse			Percentage	
	C_{off}	C_{on}	C_{enh}	ΔC	ΔC_{enh}
S1	38	26	41	−32	8
S2	139	79	131	−43	−6
S3	17	10	13	−41	−24
S4	74	33	79	−55	7
S6	5	8	8	60	60
S7	36	17	31	−53	−14
S8	18	13	21	−28	17
S9	128	56	123	−56	−4
S10	181	74	183	−59	1
S11	88	51	90	−42	2
DFG2	37	24	38	−35	3
DFG6	21	20	22	−5	5
DFG7	49	38	48	−22	−2
DFG12	86	73	85	−15	−1
DFG14	7	5	8	−29	14
DFG16	32	27	32	−16	0
DFG19	50	42	49	−16	−2
Average	**59**	**35**	**59**	**−29**	**4**

but also can determine the most appropriate reconfigurable floorplan/platform for the application.

4.4.3. Comparison with Online Schedulers. Table 7 presents a comparison between RCOnline-Enh. and two other efficient scheduling algorithms (RCSched-I and RCSched-II) that were published in [7]. Both RCSched-I and RCSched-II nominate free (i.e., inactive or currently not running) PRRs for the next ready task. If all PRRs are busy and the GPP is free, the scheduler attempts to migrate the ready task (i.e., the first task in the ready queue) by changing the task type from hardware to software. Busy PRRs, unlike free PRRs, accommodate hardware tasks that are active (i.e., running). The two algorithms differ with respect to the way the next PRR is nominated. Results obtained indicate that RCOnline-Enh. has on average 327% more hardware reuse than RCSched-I and 203% more hardware reuse than RCSched-II. Table 7 also indicates that the performance gap in terms of *total execution time* between RCSched-I and RCOnline-Enh. is, on average, 53% and that between the latter and RCSched-II is 43%.

5. Execution Unit Allocation

Optimizing the hardware architecture (alternative execution units) for a specific task in a DFG is an NP-hard problem [3]. Therefore, in this section, an efficient metaheuristic based technique is proposed to select the type of execution units (implementation variants) needed for a specific task. The proposed approach uses an *Island Based Genetic Algorithm (IBGA)* as an optimization based technique.

Table 7: Comparison of schedulers based on execution time and number of hardware reused tasks.

Benchmark	RCSched-I		RCSched-II		RCOnline-Enh.	
	Total ex. time	# of reuse	Total ex. time	# of reuse	Total ex. time	# of reuse
S1	867	10	740	19	520	41
S2	2466	35	1767	68	1160	131
S3	478	3	441	5	280	13
S4	1773	13	1508	26	660	79
S6	201	2	160	4	120	8
S7	896	6	818	10	430	31
S8	583	5	503	9	345	21
S9	2697	17	2417	44	950	123
S10	3589	22	2887	57	1075	183
S11	1735	21	1293	42	866	90
DFG2	912	18	827	22	584	38
DFG6	390	18	358	19	322	22
DFG7	701	32	596	36	436	48
DFG12	1270	51	879	66	723	85
DFG14	183	4	162	4	103	8
DFG16	469	23	403	26	288	32
DFG19	738	31	560	39	460	49
Average	**1173**	**18**	**960**	**29**	**548**	**59**

The main feature of an IBGA is that each population of individuals (i.e., set of candidate solutions) is divided into several subpopulations, called islands. All traditional genetic operators, such as crossover, mutation, and selection, are performed separately on each island. Some individuals are selected from each island and migrated to different islands periodically. In our proposed Island Based *Genetic Algorithm (GA)*, no migration is performed among the different subpopulations to preserve the solution quality of each island, since each island is based on a different (unique) floorplan as will be explained later on. The novel idea here is to aggregate the results obtained from the Pareto fronts of each island to enhance the overall solution quality. Each solution on the Pareto front tends to optimize multiple objectives including power consumption and speed. However, each island tends to optimize this multiobjective optimization problem based on a distinct platform (floorplan). The IBGA approach proposed utilizes the framework presented in [7] to evaluate the solutions created in addition to the reconfigurable simulator presented earlier in Section 3.2.

5.1. Single Island GA. The Single Island GA optimization module consists of four main components: an architecture library, Population Generation module, a GA Engine, and a Fitness Evaluation module (online scheduler), as shown in Figure 8. The *architecture library* stores all of the different available architectures (execution units) for every task type. Architectures vary in execution time, area, reconfiguration time, and power consumption. For example, a multiplier can have serial, semiparallel, or parallel implementation. Each GA module tends to optimize several criteria (objectives) based on a *single* fixed floorplan/platform.

The *initial population generator* uses a given task graph along with the architecture library to generate a diverse initial population of task graphs, as demonstrated by Figure 8. Each possible solution (chromosome) of the population assembles the same DFG with one execution unit (architecture) bound to each task. The *Genetic Algorithm Engine* manipulates the initial population by creating new solutions using several recombination operators in the form of crossover and mutation. New solutions created are evaluated and the most fit solution replaces the least fit. These iterations will continue for a specified number of generations or until no further improvement in solution quality is achieved. The fourth component, *fitness function*, evaluates each solution based on specific criteria. The fitness function used in the proposed model is based on the online reconfigurable scheduler presented earlier that schedules each task graph and returns the time and overall power consumed by the DFG to the GA Engine.

5.1.1. Initial Population. The initial population represents the solution space that the GA will use to search for the best solution. The initial population can be generated randomly or partially seeded using known solutions and it needs to be as diverse as possible. In our framework, we evaluated the system with different population sizes that varied from a population equal to the number of nodes in the (input) DFG to five times the number of nodes of the input DFGs.

Chromosome Representation. Every (solution) individual in the population is represented using a bit string known as a chromosome. Each chromosome consists of genes. In our formulation, we choose to use a gene for each of the N tasks in the task graph; each gene represents the choice of a particular implementation variant (as an integer value). A task graph is mapped to a chromosome, where each gene represents an operation within the DFG using a specific execution unit, as shown in Figure 9.

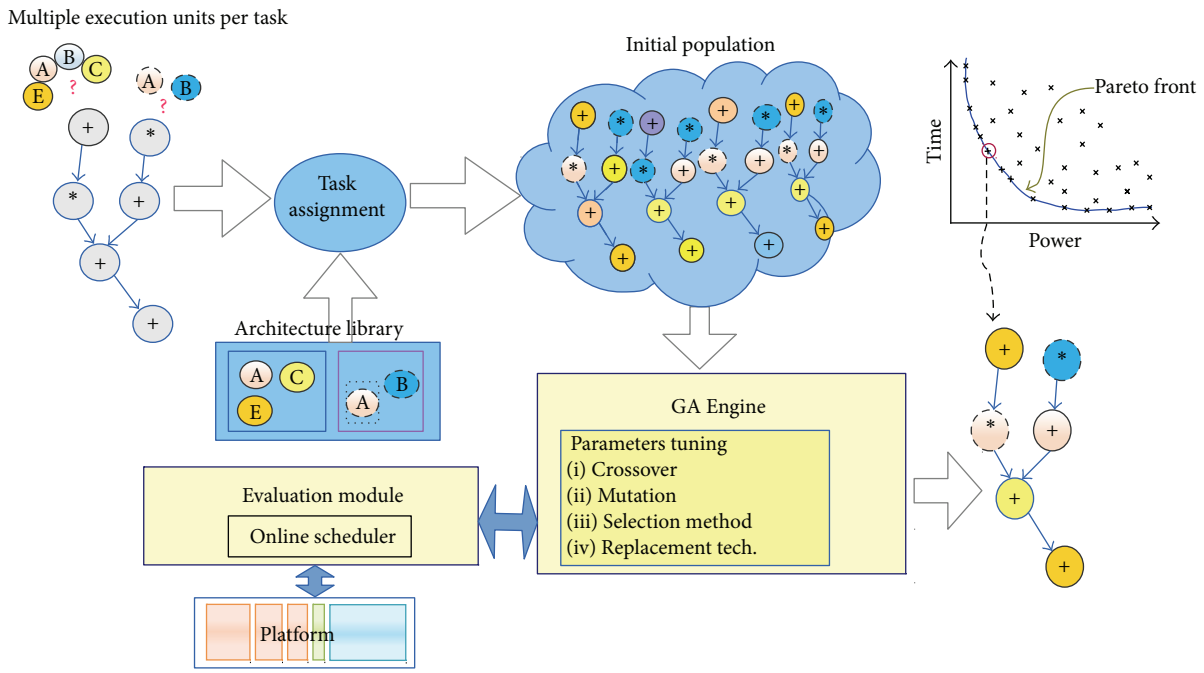


Figure 8: A Single Island GA module.

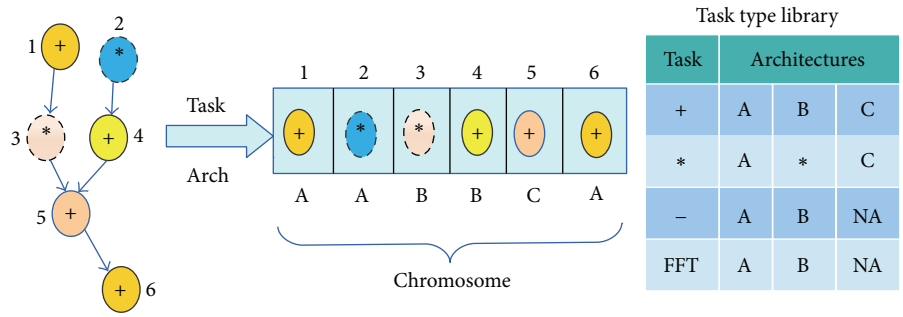


Figure 9: Task graph to chromosome mapping (binding/allocation).

5.1.2. Fitness Function. The fitness of any given chromosome is calculated and used by the Genetic Algorithm Engine as a single figure of merit. In our proposed framework, the fitness function is based on the outcome of a scheduler which takes each chromosome and uses it to construct a schedule. The scheduler then returns a value representing time, power, or a weighted average of both values. The quality of the scheduling algorithms implemented can have a significant impact on the final performance of applications running on a reconfigurable computing platform. The overall time it takes to run a set of tasks on the same platform could vary using different schedulers. Power consumption, reconfiguration time, and other factors can also be used as metrics to evaluate schedulers. The RCOnline scheduler described in Section 4.2 is used to evaluate each chromosome as part of the fitness function, in the GA framework. The results summarized in this section are based on the RCOnline scheduler presented earlier.

5.2. Island Based GA. The proposed Island Based GA consists of multiple GA modules as shown in Figure 10. Each GA module produces a Pareto front based on power and performance (execution time) for a unique platform. An aggregated Pareto front is then constructed from all the GAs to give the user a framework that optimizes not only power and performance, but also the most suitable platform (floorplan). A platform in this case includes the number, size, and layout of the PRRs.

The proposed framework simultaneously sends the same DFG to every GA island, as shown in Figure 10. The islands then run in parallel to optimize for performance and power consumption for each platform. Running multiple GAs in parallel helps in reducing processing time. The chromosomes in each island which represent an allocation of execution units (implementation variant) and binding of units to tasks are used to build a feasible near-optimal schedule. The schedule, allocation, and binding are then evaluated by the simulator to determine the associated power consumed and total execution time. These values are fed back to the GA as a measure of fitness of different solutions in the population. The GA seeks to minimize a weighted sum of performance and power consumption (2). The value of W determines the

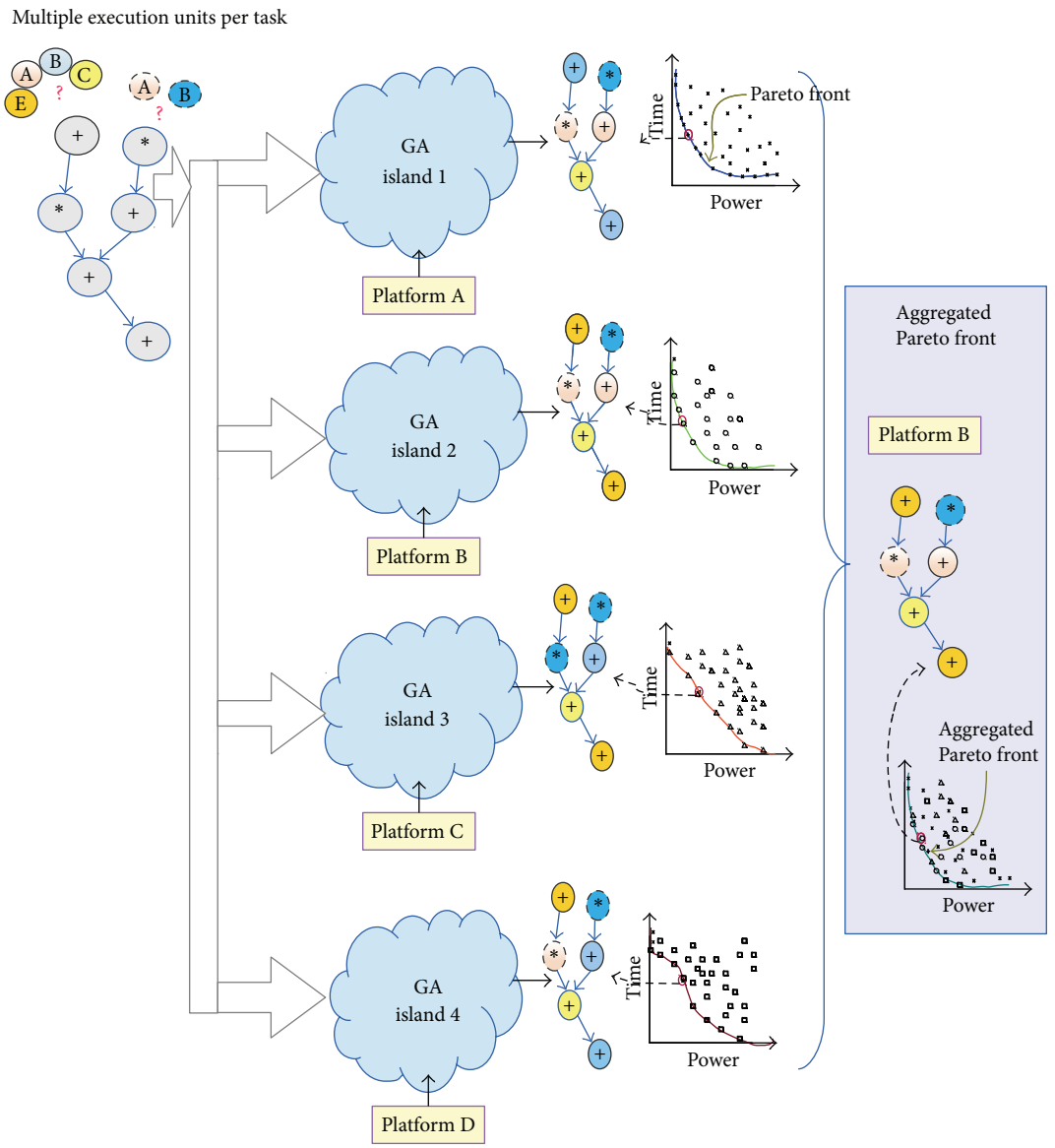


Figure 10: An Island Based GA (IBGA) framework.

weight of each objective. For example, a value of one indicates that the GA should optimize for performance only, while a value of 0.5 gives equal emphasis on performance and speed (assuming the values of speed and power are normalized):

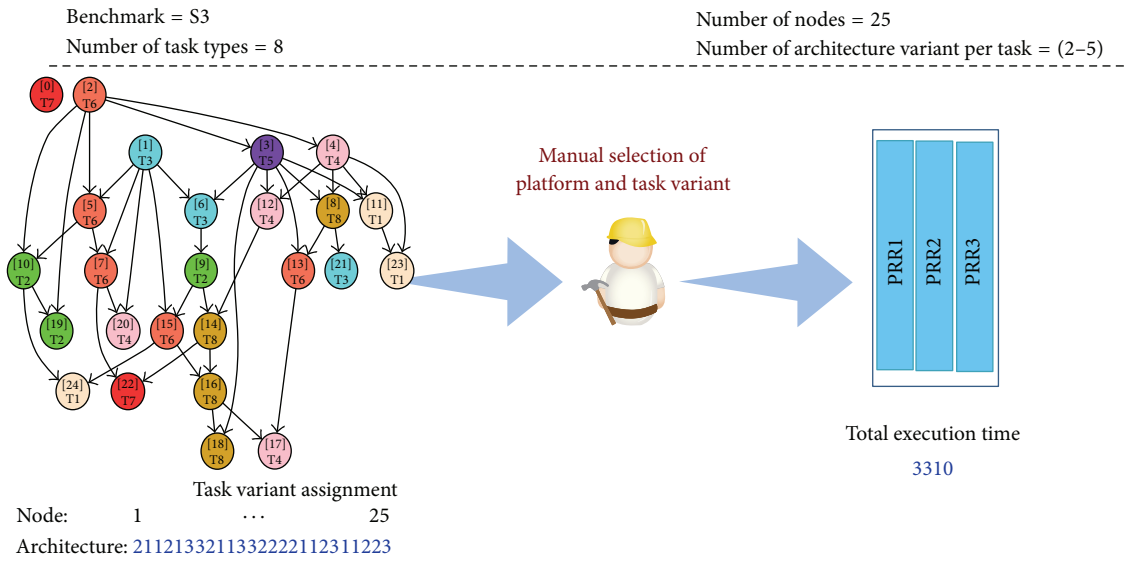
$$\text{fitness value} = \text{power} * (1 - W) + \text{exec. time} * W. \quad (2)$$

Every island produces a separate *Pareto front* for a different platform in parallel. An aggregation module then combines all the Pareto fronts into an aggregated measure that gives not only a near-optimal performance and power, but also the hardware platform (floorplan) associated with the solution. Each island generates the Pareto front by varying the value of W in (2) from 0.0 to 1.0 in step sizes of 0.1.

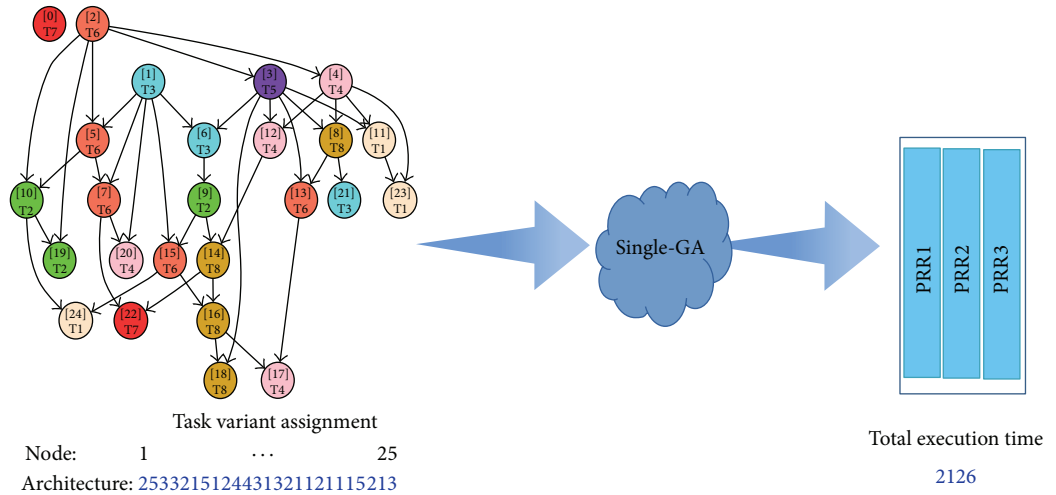
5.2.1. Power Measurements. Real-time power measurements were initially performed using the Xilinx VC707 board [35]. The Texas Instrument (TI) power controllers along with TI dongle and a monitoring station were used to perform the power measurement. Other measurements were calculated using Xilinx XPower software.

5.2.2. IBGA Example. In this section, we introduce a detailed example of the IBGA framework. Benchmark S3 will be used to illustrate how the IBGA module operates as shown in Figure 11. The S3 benchmark is a synthetic benchmark with 25 nodes and 8 different task types. Each task has between 2 and 5 different architecture variants (i.e., hardware implementation).

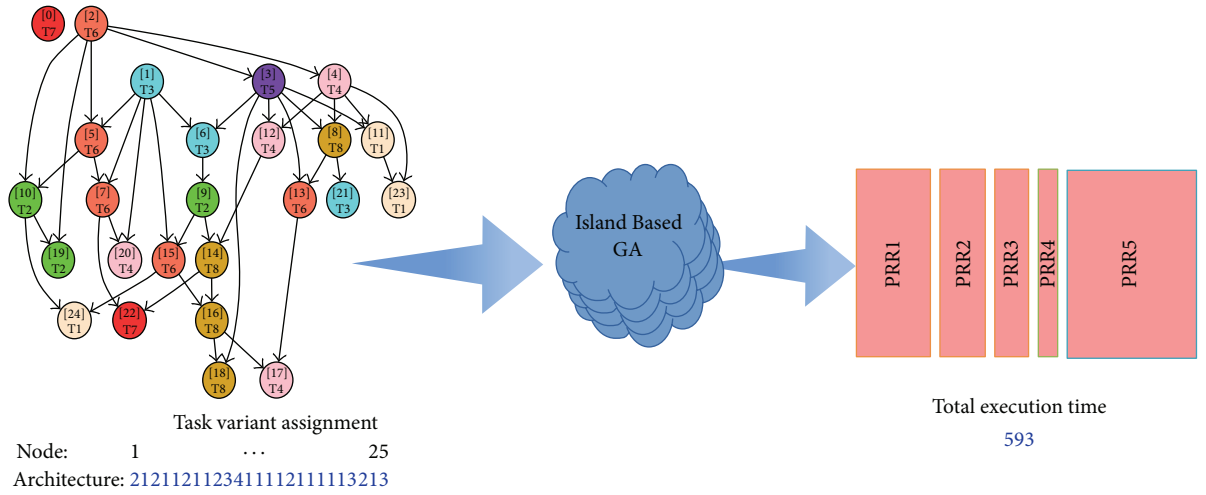
(i) *Manual Assignment.* As shown in Figure 11(a), each node is manually assigned a random hardware task variant and the platform (PRR layout) is manually selected as well. The total execution time in this case



(a)



(b)



(c)

Figure 11: IBGA example.

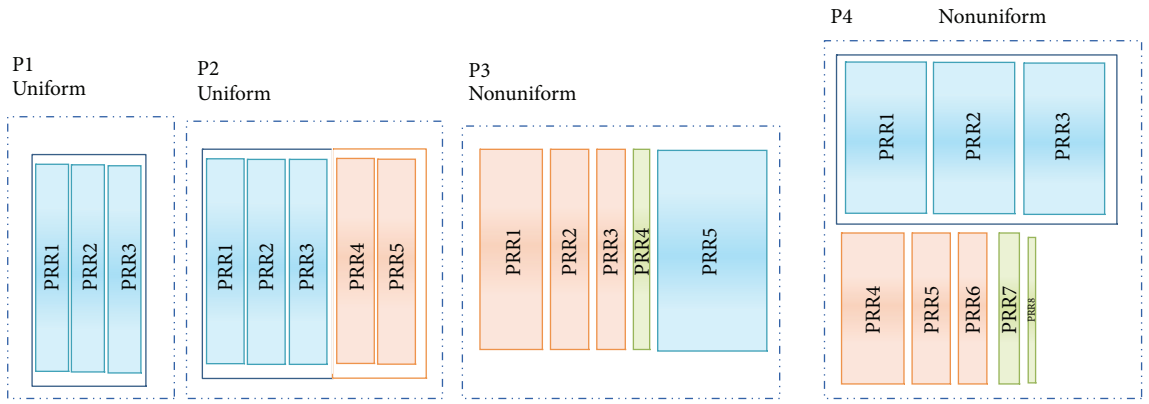


Figure 12: Platforms with different floorplans.

Table 8: Resource utilization of different multiplier implementation.

Resources	Arch 1	Arch 2	Arch 3
Slices	17	72	114
LUT	30	126	200

is 3310 cycles. The execution time will be used as a baseline for the next two cases.

(ii) *Single-GA*. Figure 11(b) shows the single-GA approach. This approach produces different task bindings than that of the manual assignment. Despite using the same platform of previous manual assignment, the different bindings lead to a more optimized DFG (lower total execution time). The performance increases by 36% over the manual task assignment.

(iii) *Island Based GA*. This approach is shown in Figure 11(c), where task binding is optimized over multiple platforms. The DFG has different task bindings than the two previous cases and uses a different platform. As a result, the performance dramatically increases (the total execution reduces from 3310 cycles to 593). The IBGA produces four different architecture bindings for each platform, as shown in Figure 12. Each number represents an index to the selected execution unit (architecture) and the location of the integer represents the node number.

Architecture binding for each platform (P1–P4), for the S3 benchmark, is as follows:

```
P1: 25332151244313211211152l3.
P2: 25332151244313211211152l3.
P3: 23222311114312111113132l2.
P4: 13222211144311111111111l2.
```

Table 8 shows the resource utilization of different multiplier implementation used in the example presented in Figure 11. Table 9 shows the resource available in a PRR and resources used by an adder in uniform implementation.

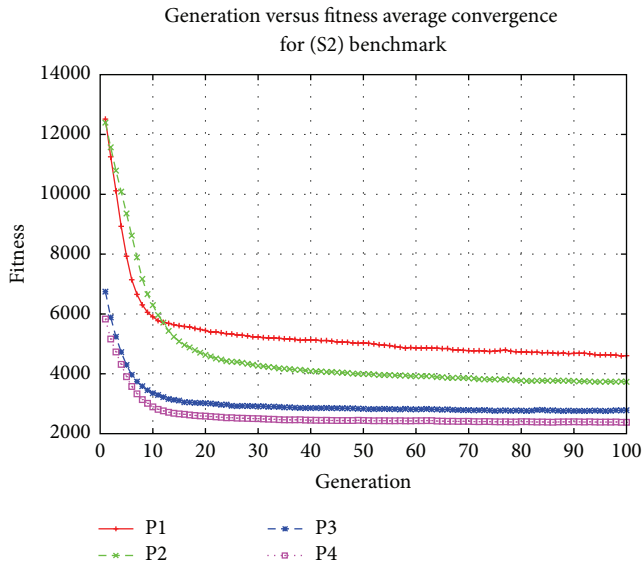


Figure 13: Convergence of synthesized benchmark S2.

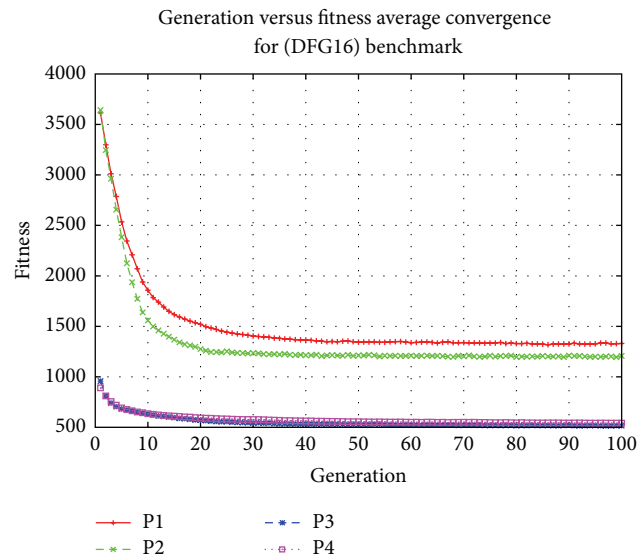


Figure 14: Convergence of MediaBench benchmark (DFG16).

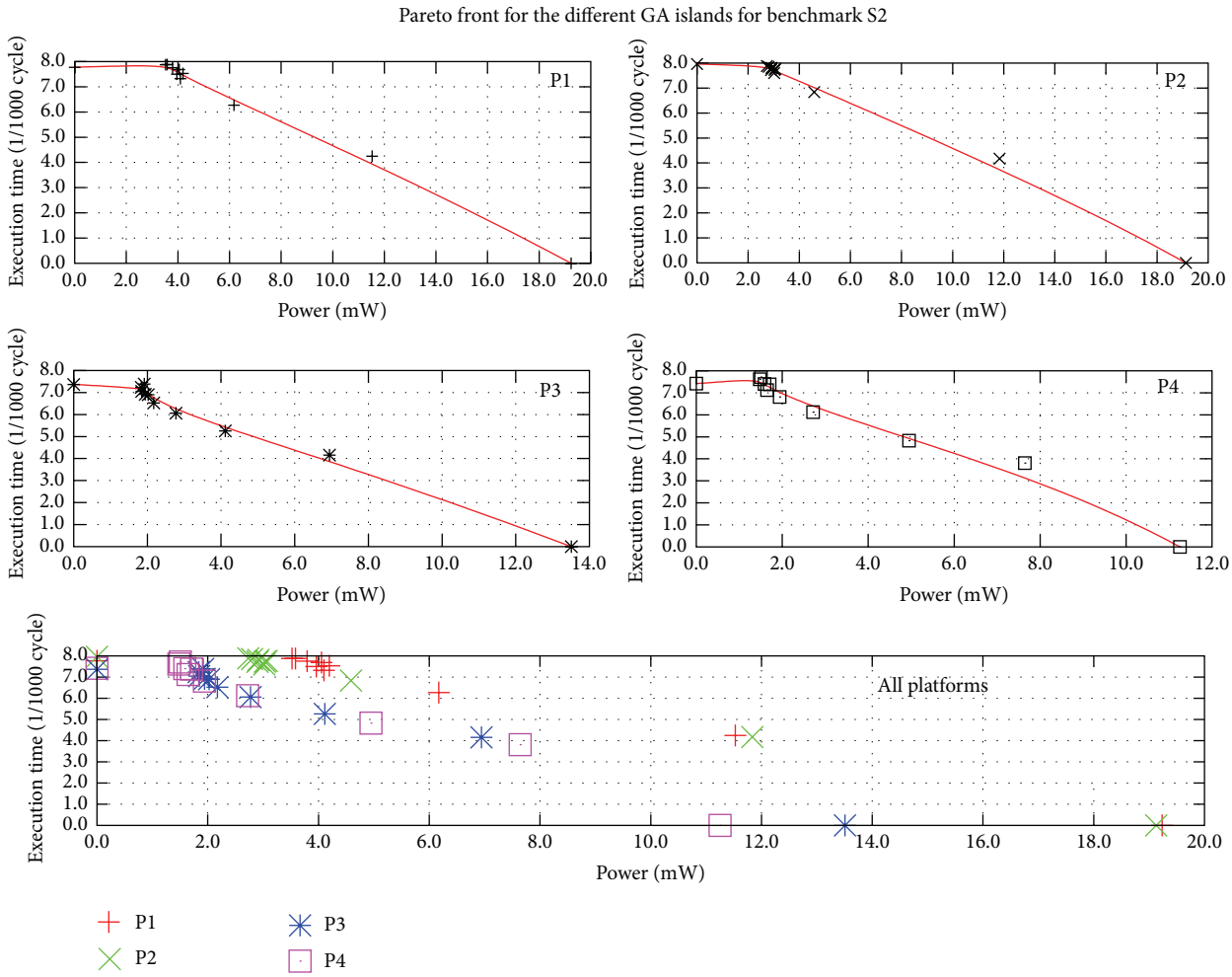


Figure 15: Aggregated Pareto front (time versus power) for synthesized benchmark (S2).

Table 9: Resources utilization of an adder in a PRR for the uniform implementation (Virtex-V).

Site type	Available	Required	Utilization %
LUT	240	32	14
FD_LD	240	32	14
SliceL	30	5	17
SliceM	30	5	17
DSP48E	4	0	0

Table 10 shows the resources available in the Virtex-V FPGA operating at 100 MHz along with the resources occupied by the complete system.

5.3. Results. In this section, we first describe our experimental setup and then present results based on the proposed Island Based GA framework. As the GA is stochastic by nature, it was run 10 times for each benchmark. Solution quality was then evaluated in terms of average results produced. The run-time performance of the IBGA based on serial and parallel implementation is shown in Table 11. The IBGA was tested on a Red Hat Linux workstation with 6-core Intel Xeon processor, running at a speed of 3.2 GHz and equipped with 16 GB of RAM.

Table 10: Resources utilization for the system on a Virtex-V FPGA (XC5VFX70T), freq. 100 MHz.

Resources	Utilization	Available	Utilization %
Register	12022	44800	26
LUT	11176	44800	24
Slice	6282	11200	56
IO	237	640	37
BSCAN	1	4	25
BUFIO	8	80	10
DCM_ADV	1	12	8
DSP48E	6	128	4
ICAP	1	2	50
Global clock buffer	6	32	18

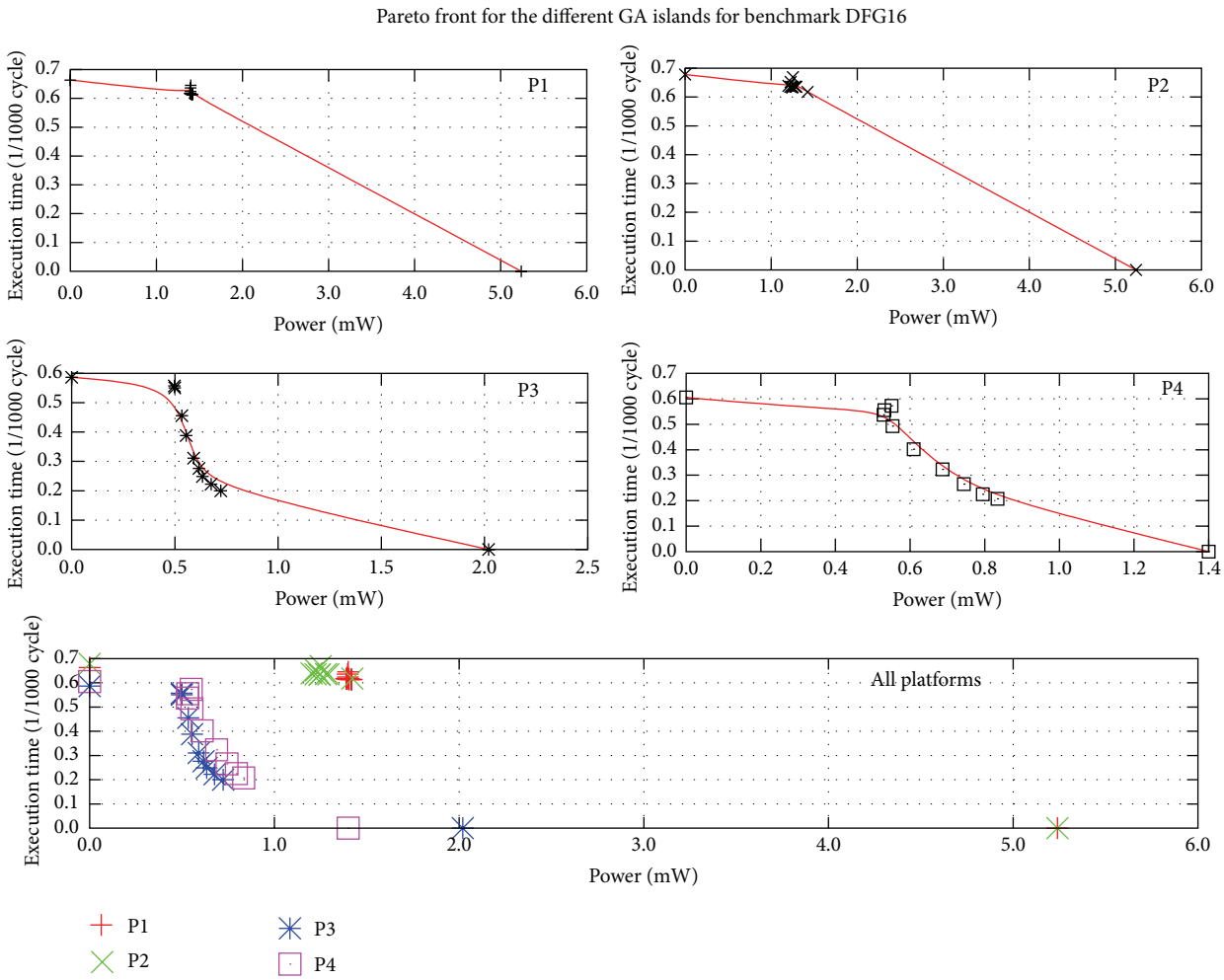


Figure 16: Aggregated Pareto front (time versus power) for MediaBench benchmark (DFG16).

Table 11: IBGA run-time for serial and parallel implementation.

DFG	Time (min:sec)		Speedup (X times)
	Serial	Parallel	
S1	00:52	00:17	3.1
S2	04:53	01:41	2.9
S3	00:23	00:10	2.3
S4	03:06	01:16	2.5
S5	00:07	00:02	3.0
DFG2	01:12	00:28	2.6
DFG6	00:12	00:08	1.5
DFG7	00:35	00:13	2.7
DFG12	02:45	00:52	3.2
DFG14	00:05	00:02	2.5
DFG16	00:26	00:09	2.9
DFG19	01:13	00:23	3.2

Table 12: The average of 10 runs for the best fitness values.

Benchmark	No opt.	1-GA	IBGA
S1	4435.7	1847	978.2
S2	12514.7	4288.3	2244.4
S3	2649.95	1973	660.9
S4	8474.23	3856	1792.4
S5	1392	877	286
DFG2	5452.67	2554.1	1036.1
DFG6	2942.99	1217.9	406.1
DFG7	4558.4	1303.2	582.4
DFG12	9249.97	2595.9	1218.1
DFG14	1134.13	735	341.5
DFG16	3614.28	1301.1	505.2
DFG19	5805.24	2368.4	1528.1

5.3.1. Convergence of Island Based GA. The benchmarks were tested on the proposed Island GA framework using four islands. Each island targeted a different FPGA platform (floorplan) that was generated manually. Each platform is distinct in terms of the number, size, and layout of the PRRs. An example is shown in Figure 12. Platforms P_1 and

Table 13: Fitness values for different weights (W).

Weight	$W = 0.5$			$W = 0.7$			$W = 0.875$			$W = 1$ (performance)		
Bench.	No opt.	1-GA	IBGA	No opt.	1-GA	IBGA	No opt.	1-GA	IBGA	No opt.	1-GA	IBGA
S1	3327	2280	1469	3917	2134	1266	4435.7	1847	978.2	4661	1601	796
S2	9483	5731	4351	11034	4872	3331	12514.7	4288.3	2244.4	13457	3724	1604
S3	1894	1517	825	2303	1760	734	2649.95	1973	660.9	2914	2126	588
S4	6286	4000	2747	7470	3890	2277	8474.23	3856	1792.4	9226	3715	1366
S5	954	684	335	1194	790	300	1392	877	286	1550	940	290
DFG2	3749	2237	1324	4673	2450	1189	5452.67	2554.1	1036.1	6025	2582	903
DFG6	1851	906	387	2434	1073	409	2942.99	1217.9	406.1	3306	1320	396
DFG7	2880	1148	668	3792	1231	640	4558.4	1303.2	582.4	5147	1324	537
DFG12	5891	2400	1432	7722	2609	1342	9249.97	2595.9	1218.1	10415	2886	1044
DFG14	775	557	338	963	656	345	1134.13	735	341.5	1249	785	340
DFG16	2274	1003	446	2976	1166	505	3614.28	1301.1	505.2	4095	1392	507
DFG19	4588	3152	2308	5232	2743	1907	5805.24	2368.4	1528.1	6146	2186	1197

P_2 have uniform size distribution, while platforms P_3 and P_4 on the other hand have nonuniform size distributions. Despite the fact that the four islands are optimizing the same task graph they converge to different solutions, since they are targeting different floorplans on the same FPGA architecture. Figures 13 and 14 show the convergence of the fitness values for two different sample benchmarks, synthetic and MediaBench, for an average of 10 runs.

5.3.2. The Pareto Front of Island GA Framework. The proposed multiobjective Island Based GA optimizes for execution time and/or power consumption. Since the objective functions are conflicting, a set of Pareto optimal solutions was generated for every benchmark, as discussed in Section 5.2.

Figures 15 and 16 show the Pareto fronts obtained for both synthetic and MediaBench benchmarks, respectively, based on an average of 10 runs. The Pareto solutions obtained by each island (P1 to P4) are displayed individually along with the aggregated Pareto front for each benchmark. It is clear from the figures that each GA island produces a different and unique Pareto front that optimizes performance and power for the targeted platform. On the other hand, the aggregated solution based on the four GA islands combines the best solutions and thus improves upon the solutions obtained by the individual GA Pareto fronts. Hence, the system/designer can choose the most appropriate platform based on the desired objective function.

Table 12 compares the best fitness values of the randomly binding architecture with no optimization (No opt.) to that using a single-GA approach (1-GA) along with the proposed aggregated Pareto optimal point approach based on four islands (IBGA). Each value in Table 12 is based on the average of 10 different runs. The single-GA implementation achieves on average 55.9% improvement over the baseline nonoptimized approach, while the Island Based GA achieves on average 80.7% improvement. The latter achieves on average 55.2% improvement over the single-GA approach. Table 13 shows the fitness values based on different weight values (introduced in (2)) for randomly binding architectures (No opt.), a single-GA (1-GA), and an Island Based GA (IBGA). On average the single-GA implementation achieves 52.7% improvement over the baseline nonoptimized approach, while the Island Based GA achieves on average 75% improvement.

Table 14: Exhaustive versus IBGA (average of 10 runs).

Benchmarks	IBGA		Exhaustive search	
	Fitness	Time (sec)	Fitness	Time (min)
DFG14 (11 nodes)	341	1.8	313	5.8
S5 (13 nodes)	286	2.4	283	257

Table 14 compares the IBGA with an exhaustive procedure in terms of quality of solution and CPU time. Only two small benchmarks are used due to the combinatorial explosion of the CPU time when using the exhaustive search based procedure. It is clear from Table 14 that the IBGA technique produces near-optimal solutions for these small benchmarks. The DFG14 (11 nodes) is 9% inferior to the optimal solution, while S5 (13 nodes) is 1% away from optimality.

6. Conclusions and Future Work

In this paper, we presented an efficient reconfigurable online scheduler that takes PRR size and other measures into account within a reconfigurable OS framework. The scheduler is evaluated and compared with a baseline reconfigurable-aware offline scheduler. The main objective of the proposed scheduler is to minimize the total execution time for the incoming task graph. The proposed scheduler supports hardware task reuse and software-to-hardware task migration. RCOnline learns about task types from previous history and its performance tends to improve overtime. RCOnline-Enh. further improves upon RCOnline and searches the ready queue for a task that can exploit hardware task reuse. RCOnline-Enh. has exceptionally good performance since on average it reached 87% of the performance of RCOffline scheduler, without the learning phase, and 104% of RCOffline scheduler following the learning phase. In addition

to the proposed schedulers, a novel parallel Island Based GA approach was designed and implemented to efficiently map execution units to task graphs for partial dynamic reconfigurable systems. Unlike previous works, our approach is multiobjective and not only seeks to optimize speed and power, but also seeks to select the best reconfigurable floorplan. Our algorithm was tested using both synthetic and real-world benchmarks. Experimental results clearly indicate the effectiveness of this approach for solving the problem where the proposed Island Based GA framework achieved on average 55.2% improvement over single-GA implementation and 80.7% improvement over a baseline random allocation and binding approach.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] K. Eguro, "Automated dynamic reconfiguration for high-performance regular expression searching," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 455–459, IEEE, Sydney, Australia, December 2009.

[2] Xilinx, *Partial Reconfiguration User Guide*, UG702, Xilinx, 2010.

[3] M. Huang, V. K. Narayana, M. Bakhouya, J. Gaber, and T. El-Ghazawi, "Efficient mapping of task graphs onto reconfigurable hardware using architectural variants," *IEEE Transactions on Computers*, vol. 61, no. 9, pp. 1354–1360, 2012.

[4] M. Huang, V. K. Narayana, and T. El-Ghazawi, "Efficient mapping of hardware tasks on reconfigurable computers using libraries of architecture variants," in *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 247–250, IEEE, Napa, Calif, USA, April 2009.

[5] S. Hauck and A. Dehon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann, Amsterdam, The Netherlands, 2008.

[6] P.-A. Hsiung, M. D. Santambrogio, and C.-H. Huang, *Reconfigurable System Design and Verification*, CRC Press, Boca Raton, Fla, USA, 2009.

[7] A. Al-Wattar, S. Areibi, and F. Saffih, "Efficient on-line hardware/software task scheduling for dynamic run-time reconfigurable systems," in *Proceedings of the IEEE Reconfigurable Architectures Workshop (RAW '12)*, pp. 401–406, IEEE, Shanghai, China, May 2012.

[8] A. Al-Wattar, S. Areibi, and G. Grewal, "Efficient mapping and allocation of execution units to task graphs using an evolutionary framework," in *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '15)*, Boston, Mass, USA, June 2015.

[9] A. Al-wattar, S. Areibi, and G. Grewal, "Rcsimulator, a simulator for reconfigurable operating systems," 2015, https://github.com/Aalwattar/rcSimulator.

[10] "Island based genetic algorithm for task allocation," 2015, https://github.com/Aalwattar/GA-Allocation.

[11] H. Walder and M. Platzner, "A runtime environment for reconfigurable hardware operating systems," in *Field Programmable Logic and Application*, pp. 831–835, Springer, Berlin, Germany, 2004.

[12] "Reconfigurable hardware operating systems: from design concepts to realizations," in *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA '03)*, pp. 284–287, CSREA Press, 2003.

[13] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[14] F. Ghaffari, B. Miramond, and F. Verdier, "Run-time HW/SW scheduling of data flow applications on reconfigurable architectures," *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 976296, p. 3, 2009.

[15] J. A. Clemente, C. González, J. Resano, and D. Mozos, "A task graph execution manager for reconfigurable multi-tasking systems," *Microprocessors and Microsystems*, vol. 34, no. 2–4, pp. 73–83, 2010.

[16] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, and J. Becker, "Operating system for runtime reconfigurable multiprocessor systems," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 121353, 16 pages, 2011.

[17] D. Göhringer and J. Becker, "High performance reconfigurable multi-processor-based computing on FPGAs," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–4, IEEE, Atlanta, Ga, USA, April 2010.

[18] T.-M. Lee, J. Henkel, and W. Wolf, "Dynamic runtime rescheduling allowing multiple implementations of a task for platform-based designs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 296–301, Paris, France, March 2002.

[19] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 149–158, IEEE, April 2005.

[20] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the Workshop on Circuits Systems and Signal Processing (ProRISC '00)*, Veldhoven, The Netherlands, November-December 2000.

[21] J. Wang and S. M. Loo, "Case study of finite resource optimization in fpga using genetic algorithm," *International Journal of Computer Applications*, vol. 17, no. 2, pp. 95–101, 2010.

[22] Y. Qu, J.-P. Soininen, and J. Nurmi, "A genetic algorithm for scheduling tasks onto dynamically reconfigurable hardware," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 161–164, Napa, Calif, USA, May 2007.

[23] R. Chen, P. R. Lewis, and X. Yao, "Temperature management for heterogeneous multi-core FPGAs using adaptive evolutionary multi-objective approaches," in *Proceedings of the IEEE International Conference on Evolvable Systems (ICES '14)*, pp. 101–108, IEEE, Orlando, Fla, USA, December 2014.

[24] A. Elhossini, S. Areibi, and R. Dony, "Strength pareto particle swarm optimization and hybrid EA-PSO for multi-objective optimization," *Evolutionary Computation*, vol. 18, no. 1, pp. 127–156, 2010.

[25] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10*, pp.

60.1–60.10, Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), Brussels, Belgium, 2010.

[26] U. Electrical & Computer Engineering Department at the UCSB, "Express benchmarks," 2015, http://express.ece.ucsb.edu/benchmark/.

[27] A. Bilgin and J. Ellson, "Dot (graph description language)," 2008, http://www.graphviz.org/Documentation.php.

[28] G. A. Vera, D. Llamocca, M. S. Pattichis, and J. Lyke, "A dynamically reconfigurable computing model for video processing applications," in *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, pp. 327–331, IEEE, Pacific Grove, Calif, USA, November 2009.

[29] L. Cai, S. Huang, Z. Lou, and H. Peng, "Measurement method of the system reconfigurability," *Journal of Communications and Information Sciences*, vol. 3, no. 3, pp. 1–13, 2013.

[30] J. A. Clemente, J. Resano, C. González, and D. Mozos, "A Hardware implementation of a run-time scheduler for reconfigurable systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 7, pp. 1263–1276, 2011.

[31] C. Steiger, H. Walder, and M. Platzner, "Heuristics for online scheduling real-time tasks to partially reconfigurable devices," in *Field-Programmable Logic and Applications: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1–3, 2003: Proceedings*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 575–584, Springer, Berlin, Germany, 2003.

[32] Y.-H. Chen and P.-A. Hsiung, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC," in *Embedded and Ubiquitous Computing—EUC 2005*, vol. 3824, pp. 489–498, Springer, Berlin, Germany, 2005.

[33] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi dimensional reconfigurable architectures," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 97–102, Cancun, Mexico, December 2008.

[34] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 519–522, Munich, Germany, March 2008.

[35] Xilinx Inc, *Xpower Estimator User Guide*, Xilinx Inc, 2010.