IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 10 August 2022, accepted 21 August 2022, date of publication 6 September 2022, date of current version 19 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3204866

APPLIED RESEARCH

Virtualizing and Scheduling FPGA Resources in Cloud Computing Datacenters

ABID FARHAN, RAAFAT ABURUKBA, ASSIM SAGAHYROON, (Senior Member, IEEE), MOHAMMED ELNAWAWY, AND KHALED EL-FAKIH

Department of Computer Science and Engineering, American University of Sharjah, Sharjah, United Arab Emirates

Corresponding author: Raafat Aburukba (raburukba@aus.edu)

This work was supported in part by the Department of Computer Science and Engineering, American University of Sharjah; and in part by the Open Access Program from the American University of Sharjah.

ABSTRACT Cloud service providers consistently leverage their computing infrastructures by adding reconfigurable hardware platforms such as field-programmable gate arrays (FPGAs) to their existing infrastructures. Adding FPGAs to a cloud environment involves non-trivial challenges. The first challenge is virtualizing FPGAs as part of the cloud resources. As a standard virtualization framework is lacking, there is a need for an efficient framework for virtualizing FPGAs. Furthermore, FPGA resources are used in conjunction with central processing units (CPUs) and graphics processing units (GPUs) to accelerate the execution of tasks. Therefore, to gain the benefits of these powerful accelerating platforms, the second challenge is to optimize the allocation of tasks into the capable resources within a cloud data center. This work proposes an FPGA virtualization framework that abstracts the physical FPGAs into virtual pools of FPGA resources. The work further presents an integer linear programming (ILP) optimization model to minimize the makespan of tasks where FPGA resources are part of the cloud data center. Given the complex nature of the problem, a simulated annealing (SA) metaheuristic is developed to achieve gains in performance compared to the exact method and to scale up and handle many tasks and resources while providing near-optimal solutions. Experimental results show that SA has reduced the makespan of a large dataset with 1000 tasks and 100 resources by up to 30% when compared to first-come-first-served (FCFS) and shortest-deadline-first (SDF) algorithms. Lastly, to quantify the performance of FPGA-enabled cloud datacenters, the work extends the CloudSim simulator (an open-source cloud simulator) to enable FPGA as a resource in its environment. The proposed virtualization framework and the SA scheduler are integrated into the environment. Simulation results show that the execution time of tasks is reduced by up to 78% when FPGA accelerators are used.

INDEX TERMS FPGA, cloud computing, virtualization, scheduling, CloudSim, simulated annealing.

I. INTRODUCTION

Cloud computing is one of the active research areas in computing. Recently, hardware accelerators have been introduced in cloud datacenters. Along with traditional resources, accelerators must be integrated carefully, which involves solving several non-trivial challenges. Such hardware platforms speed up cloud services and applications tremendously compared to conventional platforms. The National Institute of Standards and Technology (NIST) [1], defined cloud computing as a model for enabling abundant, convenient, network access on user demand to a shared pool of configurable computing resources. These resources mainly include servers, storage devices, and networks, among other entities. They can be provisioned rapidly, with minimal management effort and human interaction. In addition, virtualization is one of the main enabling technology in cloud computing. Virtualization provides a logical abstraction of the physical resource so that one single computer can run multiple operating systems [2].

This research focuses on enabling hardware acceleration services in a cloud data center. The notion of

The associate editor coordinating the review of this manuscript and approving it for publication was Nitin Gupta.

Accelerator-as-a-Service (AaaS) is relatively new in cloud computing, especially using FPGA devices. Clouds offering AaaS allow users to request various acceleration services without requiring any technical knowledge of the accelerator hardware. All hardware management and configurations carried out by the cloud provider are hidden from users.

To enable AaaS, one must consider possible approaches to FPGA virtualization. According to [3], there are four types of virtualizations: Single FPGA Single Application (SFSA), Single FPGA Multiple Applications (SFMA), Multiple FPGAs Single Application (MFSA), and Multiple FPGAs Multiple Applications (MFMA). A modern mechanism called dynamic partial reconfiguration (DPR) is used for multiple applications on a single device. The FPGA fabric is logically partitioned into multiple regions, usually symmetric, which can be used to configure application hardware. The term ''dynamic'' implies that an FPGA is already running some configured tasks that can be reconfigured to accommodate additional tasks at runtime.

On the other hand, the term ''partial'' is used because specific regions can be reconfigured for application hardware while the rest of the regions in the FPGA are untouched. Many modern FPGAs that support DPR and vendor-specific tools are capable of being portioned where static logic or shell is required. The static logic never changes once configured in the FPGA. Furthermore, the static logic must define how to handle the different data coming in and going out of the FPGA. Moreover, static logic modules must manage application or user data and establish well-defined communication protocols between themselves and accelerators (i.e., user hardware). Application logic modules, which are accelerator hardware designs, are configured in partially reconfigurable regions (PRRs). The term ''partial'' implies that only some portion of the FPGA gets its resource elements reconfigured for the new application logic to be implemented. A PRR is also referred to as a *role*. Hence, a shell is never modified once configured, but roles may be reconfigured as often as required. It also allows multiple applications to run on FPGA, enabling resource sharing and multitenancy where multiple users can share the same physical resource. In the cloud datacenter context, the Multiple FPGAs Multiple Applications (MFMA) virtualization type is the ideal choice of virtualization.

The research work in this paper has the following contributions:

- Proposes a virtualization framework for FPGA resources in a cloud data center. The proposed framework is modular and allows high resource utilization. As a result, we obtain several partitions or regions from each physical FPGA and aggregate them to form resource pools.
- Develops an optimization scheduling model that minimizes the makespan in the cloud to the virtualized pool of FPGA resources.
- Implements and evaluates the simulated annealing algorithm that obtains a near-optimal solution for the modeled scheduling problem.
- Extends the CloudSim simulation toolkit to include the proposed FPGA virtualization framework and scheduling algorithm to validate the proposed solution.

The rest of the paper is organized as follows: Section II discusses the research work in the literature. Section III proposes the FPGA virtualization framework. Section IV discusses the proposed model for FPGA resource scheduling. Section V describes the proposed heuristic-based scheduling algorithm. Section VI explains the various experiments conducted and the obtained results. Lastly, Section VII concludes the paper.

II. RELATED WORK

Multiple research approaches in the literature discuss virtualization, partition styles, and resource allocation for FPGA within different settings. This section reviews the related work and discusses the research gap that this work aims to solve.

A. FPGA VIRTUALIZATION APPROACHES

Chen *et al.* [4] proposed a virtualization framework for enabling FPGAs in the cloud. The hardware layer of the framework is divided into three logical sublayers – user sublayer, service sublayer, and platform sublayer. These layers have static hardware modules implemented within the FPGA fabric. The platform sublayer consists of static functional components such as memory and network controllers that handle data communication from and to the FPGA. The service sublayer manages the configuration of application hardware logic and data using modules such as a configuration controller, a job queue, and a job scheduler. This is considered the most significant layer because it enables partial reconfiguration of accelerators and provides interfaces for users to access their FPGA accelerators. The user sublayer comprises a static layout with four asymmetric partitions in the topmost sublayer. The partitions, also known as empty accelerator slots or partially reconfigurable regions (PRRs), are marked with alphabets A, B, C, and D. When the configuration controller receives the hardware application logic in the form of bitstreams, it configures the hardware defined by that logic into one of the four accelerator slots. Moreover, the framework has a hypervisor layer which is responsible for receiving user requests to create accelerators. Using the Accelerator-as-a-Service (AaaS) model, when a user requests for a specific accelerator, the hypervisor layer either selects an idle accelerator slot to configure the requested accelerator or finds an existing accelerator belonging to the user, or rejects the request if there is no slot available.

In addition, the hypervisor tracks the usage and status of each accelerator slot, whether the slot is idle or occupied. Using this framework, each FPGA is divided into four on-chip accelerator slots, and each slot is different in terms of the number of resources constrained by the logical partitioning.

Implementing this virtualization framework across several FPGAs leads to obtaining four distinct pools of accelerator slots. Each pool contains only one type of slot, i.e., either A, B, C, or D. The partitioning as well as implementation details of the framework are hidden from the cloud users. However, there is a significant disadvantage in using a static layout of asymmetric partitions because it renders the abstraction scheme inefficiently. Despite multiple accelerators being configured on a single FPGA chip (SFMA virtualization), using this framework, a single accelerator cannot be configured across multiple FPGAs (MFSA virtualization). Moreover, suppose one type of accelerator is dominantly requested. In that case, each FPGA has only the corresponding slot utilized, leaving the rest of the slots idle, and as a result, it would lead to poor resource utilization. Lastly, accelerator migration, which is a core feature of clouds, is not supported.

Microsoft proposed a virtualization scheme in [5] and [6] where they implemented the Bing search engine across several FPGAs, obtaining a very high throughput. The work virtualized the FPGA resources at the network, which contains FPGAs and host machines. As a result, FPGAs cannot run multiple applications on the chip, but a single application hardware can be configured across multiple FPGAs (MFSA type). The drawback is that small-scale applications that might require only a portion of an FPGA would occupy an entire chip, leading to poor resource utilization. Moreover, accelerator migration is overly complex. Hence this framework is not effective in a cloud infrastructure.

In [7], the authors proposed an operating system (OS) for FPGAs called Feniks that uses the static logic of Microsoft's Catapult in [6]. The static regions, where hardware such as memory and network controllers are configured, are known as OS regions. Moreover, the PRRs, known as application regions, are a result of partitioning the FPGA fabric. In addition, the virtualization framework provides the logical abstraction of the physical PCIe (short for peripheral component interconnect express) interface, which establishes an efficient communication channel between server resources such as storage and network devices and the FPGA chip. As a result, each application region can access local and cloud resources through the PCIe and Ethernet interfaces, respectively. Moreover, Feniks supports both MFSA and MFMA virtualization types; therefore, it is applicable in cloud infrastructures. In addition, a near-identical framework is proposed in [8].

Al-Aghbari and Elrabaa [9] proposed a scheme that supports MFMA virtualization. It performs resource-level virtualization in FPGA's I/O channels and uses network-attached FPGAs. Moreover, it has static hardware modules that perform specific logical functions, for example, a network controller external to the FPGA fabric that manages Ethernet-based communication. In addition, the framework has a reconfiguration manager that safely configures accelerators in the PRRs, which are referred to as virtual FPGAs or vFPGAs. A vFPGA uses a static interface to communicate with the static hardware modules. This interface, called the wrapper, is automatically generated based on the user's specifications. The wrapper virtualizes the physical I/O resources in the FPGA and allows users to design accelerator hardware without considering physical I/O constraints. Furthermore, Al-Aghbari *et al.* implemented their framework in real-world cloud infrastructure [10]. They explained the implementation of the FPGA hypervisor and elaborated on how its frontend functions are exposed to the cloud user as application programming interfaces (APIs) while the backend functions are implemented in the FPGA chip. Using the hypervisor, users can create, manage, and destroy accelerators configured in the vFPGAs. Moreover, each accelerator in the vFPGA is assigned an IP address allowing any host machine or FPGA in the network to communicate with the accelerator. Additionally, the authors used Xilinx Virtex-6 XC6VLX550T FPGAs as their hardware platform. They further implemented their virtualization framework and used 58,123 look-up tables, 52,649 flip-flops, 422 random access memory (RAM) blocks, and 560 digital signal processing (DSP) blocks in each FPGA.

B. FPGA PARTITIONING STYLES

DPR-supported FPGAs perform reconfigurations based on a partitioning style. Partitioning refers to the way partially reconfigurable regions (PRRs) are formed after logically dividing the FPGA fabric. According to [11], there are three partitioning styles – island style, slot style, and grid style. The island-style partitioning is the least difficult to implement where an FPGA has one or more PRRs. Each PRR can exclusively configure one application hardware, and an application cannot share more than one PRR, hence the name ''island.'' However, the limitation to this partitioning style is when a single island does not have sufficient resources to configure the hardware. The slot style partitioning is where identical PRRs called slots are created on the fabric, either column-wise or row-wise. Hardware designs then occupy one or more slots once they are configured. This partitioning is not straightforward to implement because each slot must have a static interface to communicate between the configured application hardware and the static hardware. However, it addresses the limitation of the previous partitioning style. Hence, an application can be provisioned for more than one slot if required. Another challenge in this partitioning style is that an FPGA has heterogeneous resources. Therefore obtaining identical partitions or slots that are architecturally homogenous is challenging. Since some regions in the FPGA fabric are excluded from being a part of any slot, the resources in these regions are unutilized, leading to poor resource utilization. Finally, the grid style partitioning is where an FPGA is segmented into a grid, and the grid cells are either static regions or PRRs, depending on the type of hardware configured. Hardware designs occupy one or more cells to actualize accelerators. Moreover, it improves resource utilization compared to the slot style because fewer unpartitioned regions exist. In contrast, implementing this style and producing homogenous grid cells is more challenging as an FPGA fabric contains heterogeneous resources. In addition, each PRR cell must

have a static communication interface that must be carefully designed to occupy as few resources as possible and allow more room for application hardware logic.

C. FPGA SCHEDULING APPROACHES IN CLOUD

Scheduling is the process of allocating resources to a set of tasks at a specific time [12]. In cloud, resources are virtual resource pools that are either homogenous, i.e., have only one type of resource, or heterogeneous, depending on the virtualization framework used. Moreover, examples of tasks in a cloud include a user requesting a VM for general-purpose processing, deploying a custom software application to be hosted, and using hardware acceleration for image processing. Additionally, scheduling is always carried out to fulfill an objective: either a minimizing function, e.g., power consumption of computer systems and makespan of tasks, or a maximizing function, e.g., resource utilization of network bandwidth during data transmission and memory utilization. Note that makespan refers to the completion time for a set of tasks. Furthermore, scheduling may be performed with task constraints and/or resource constraints, and the inability to achieve a constraint makes a candidate solution infeasible.

There are two classes of scheduling problems – static and dynamic. In static scheduling, all information, such as the resources capabilities and tasks requirements, are known to the scheduler. On the other hand, dynamic scheduling refers to changes within the environment that must be known to the scheduler to provide a feasible solution. Such changes are: resource failure, introducing a new resource, and new task arrivals. Cloud infrastructures typically use dynamic scheduling since tasks arrive in real-time, and the scheduler must allocate resources for these tasks without prior knowledge. In addition, many scheduling algorithms exist that can be categorized into either exact solution methods or heuristic-based methods. Exact methods always guarantee the best or the optimal solution for a scheduling problem. However, they are unable to scale when tasks grow either in quantity or complexity. Hence, scalable approaches such as heuristic-based algorithms that provide a solution within an acceptable time are needed. Heuristic approaches do not guarantee optimality. However, heuristic algorithms can scale to large size problems and provide a near-optimal solution with better performance.

Some traditional task allocation approaches in cloud infrastructures for resources are the First-Come-First-Serve, Round-Robin, Priority- or Metric-Based, and Approximation-Based. Others looked into heuristic-based methods such as Swarm Intelligence, Genetic Algorithm, and Simulated Annealing [12], [13], [14], [15], [16], [17], [18]. After introducing FPGAs as cloud resources, many implementations of these algorithms are being proposed in the literature, making FPGA scheduling an active research topic. For example, the proposed scheduling algorithm in [19] is based on the virtualization framework of Chen *et al.* in [4]. Accelerator slots (PRRs) of various sizes are obtained because of the virtualization and the resources that the proposed scheduler provisions. In the scheduling algorithm, an accelerator slot's computing capacity is a parameter defined as the number of virtual CPUs (vCPUs). Suppose a task is executed using one of the accelerator slots and accelerated n times faster than a single vCPU. In that case, the slot's computing capacity is said to be n vCPUs. Moreover, it is a metric-based scheduling algorithm where the proposed metric is called benefit. The benefit of an accelerator slot is calculated by summing the speedup of all tasks on that accelerator slot in terms of number of vCPUs. The objective is to assign each task to the slot with the highest benefit value for that task. Additionally, the authors make the scheduling algorithm dynamic by allowing task preemption where new incoming tasks yielding higher benefit on a particular slot can replace the existing ones. However, there are two disadvantages in the algorithm. The first is that the resource pools are based on the virtualization framework of [4] and consist of heterogenous partitions or accelerator slots. This leads to poor resource utilization as bigger accelerator slots will always hold higher benefit value than the smaller ones, and the scheduler always selects slots with the highest benefit for every task. The second issue is that the scheduler must calculate the benefit across the entire pool of accelerator slots for each task before resource allocation. This is computationally intensive and therefore, the algorithm is not scalable with larger number of resources or tasks.

In [20], an FPGA resource scheduling algorithm is proposed that minimizes the makespan of a batch of requests to improve resource utilization at the node level. The requests are acceleration tasks that are to be executed using FPGA hardware. They are split into three categories – computation-intensive, network-intensive, and a combination of both. Three optimization models are presented to tackle each task category independently. These models represent NP-hard problems, and as a result, an approximation algorithm is proposed that uses relaxation and rounding to find feasible solutions. The results of the algorithm are compared to shortest-job-first and longest-job-first algorithms. However, the proposed models work with a resource pool that constitutes whole physical FPGA chips instead of PRRs. Therefore, the algorithm only allocates one or more chips per task and cannot allocate low-level FPGA resources, resulting in poor resource utilization.

Most of the proposed scheduling algorithms in the literature consider the whole FPGA chips within the resource pool. The authors in [21] proposed a metric-based multi-objective scheduler that minimizes the energy consumption by allocating computation-intensive tasks to compute nodes with FPGAs. The scheduler decides whether to schedule a task to a compute node with or without an FPGA based on the tasks' workload. Moreover, in [22], the proposed model is a max-min joint optimization model which maximizes cloud users' satisfaction while minimizing loss of benefits for the cloud providers. The proposed algorithm is a generic MATLAB scheduler presented as a black box. Although the authors claim that the resource pool was obtained because

of MFMA virtualization, this is not indicated as the pool contains whole FPGA chips instead of slots or PRRs.

It was observed from the reviewed scheduling algorithms that most of the algorithms deal with resources from a resource pool consisting of whole FPGA chips [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. Some works do not virtualize the FPGAs; instead, physical chips are allocated to tasks. Such schedulers do not allow configuring more than one application per FPGA chip and do not complement MFMA virtualization frameworks. In Section IV, we formulate an ILP optimization model to schedule tasks using a resource pool of PRRs. In Section VI, we propose a scheduling algorithm based on the proposed model to perform task scheduling.

III. PROPOSED FPGA VIRTUALIZATION FRAMEWORK

This section presents our proposed approach that abstracts the physical FPGA resources into logical virtual resources. The characteristics of the FPGA resources are extracted and analyzed to build the virtualization framework. As a result, an FPGA resource pool is obtained that contains FPGA resources in the form of PRRs. Moreover, our simulation framework is based on the hardware platform developed by [9] and [10].

A. FPGA CHARACTERISTICS OVERVIEW

Characterizing an FPGA chip yields the following relevant attributes: configurable logic blocks (CLBs) within a cloud environment, which are the essential building blocks of any circuit on an FPGA, DSP slices for fast arithmetic processing, blocked RAMs (BRAM) memory resources, clocks, I/O blocks, and transceivers. Sophisticated modules such as network manager, clock manager, configuration manager, and ICAP (Internet Content Adaptation Protocol) interface are also included.

The virtualization framework uses a dynamic partial reconfiguration (DPR) mechanism. The framework focuses on abstracting the physical connectivity to the FPGA resources to enforce flexibility, standard protocols to reduce implementation complexity, and advanced partitioning to increase resource utilization. For the logical abstraction of physical connections, the PCIe interface of the FPGA connects it to a host server, while the Ethernet interface of the same FPGA is used to connect to a network switch. This allows the FPGA to be used as a local and remote accelerator. For partitioning, grid-style partitioning is used to obtain architecturally homogenous regions. Since the considered FPGAs in this work assumes DPR support, the dynamic regions are reconfigured via the ICAP interface with user designs. Furthermore, the configuration manager implements the desired FPGA accelerators. However, the architecture of FPGA comprises heterogeneous resources. Therefore, it is impossible to partition the fabric's entirety into a set of homogenous regions [31].

Fig. 1 shows an overview of the virtualization framework. The key components of the framework are the network manager, the clock manager, the configuration manager, and the vFPGA manager. The following sections elaborate on each manager with respect to their operations and dependencies with other components.

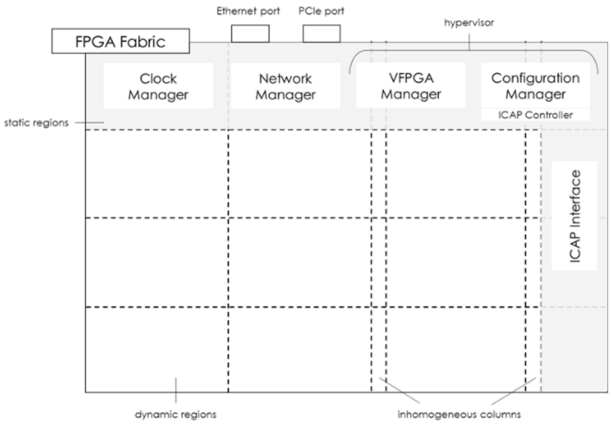

FIGURE 1. FPGA virtualization framework overview.

B. CLOCK MANAGER

One of the static modules is the clock management. It is also known as the clock manager. It is possible to have different modules run on different clock frequencies in the same FPGA. For example, the network manager may be operating at a clock frequency different from the configuration manager. Such modules are then said to be operating in different clock domains. The clock manager enables the correct frequency clock signals to be routed to both synchronous modules. Furthermore, the clock manager can utilize PLLs (phase-locked loops) available in the FPGA to generate the different clock frequencies needed. Although the primary function of a PLL is to detect and fix time violations in the circuit, it can also be used as a clock generator to drive clock signals at the desired frequency to one or more modules of the same clock domain. However, an extra piece of hardware is required when communication between two clock domains operates at different speeds. We use asynchronous first-in-first-out (FIFO) buffers extensively for inter-clock domain communication. When a module transfers data in a specific frequency to another module, it writes the signal or the data to a buffer. The recipient module can then read from the asynchronous buffer at its operating frequency. Using buffers eliminates the danger of metastability – a state in which a digital system becomes unstable and gives an uncertain output (i.e., neither logical '1' nor '0') for an unbounded time [32].

C. NETWORK MANAGER

The network manager oversees all communication external to the FPGA. Fig. 2 illustrates how the architecture of this module allows it to receive and send packets via Ethernet and PCIe ports. A packet arrives at the physical receiver through either of the two ports. It is decapsulated to extract only useful

information, such as the payload. If the packet comes through the Ethernet port, it is then an Ethernet packet and contains an IP header since an Ethernet connection uses TCP/IP protocol. However, if the packet arrives via PCIe port, we still implement the TCP/IP stack in the PCIe communication and ensure that the sender has put a destination IP address into the packet. This is the major difference in the implementation from [33], where authors used direct memory access (DMA), which is typical in a PCIe-attached FPGA infrastructure. However, by eliminating DMA and instead using TCP/IP protocol for data packets in PCIe, we reduce the complexity of establishing communication between the host and FPGA. In addition, since Ethernet packets also use TCP/IP stack, a single implementation for sending and receiving data from both the ports in the network manager is applied.

As a packet is received and decapsulated, the payload is sent to a data router, whereas the destination IP address from the IP header is sent to the IP address table for cross-checking. The IP address table contains 1-to-1 mapping of each accelerator's IP address and the vFPGA manager's physical address. This module manages and monitors all accelerators in the FPGA. If a match is found corresponding to one of the accelerators, the payload is intended for that accelerator. Therefore, the router forwards the payload, ensuring it reaches the right destination via the vFPGA manager. The vFPGA manager is discussed in detail in section G.

On the other hand, if the payload contains bit files and is intended for the configuration manager, the destination IP address will be the configuration manager. This entails that the configuration manager module in the FPGA is assigned a unique IP address for itself so that packets can be sent accordingly. Once the router gets a match of the IP address from the address table, it routes the payload to the configuration manager. Any routing from the data router to either the vFPGA manager or the configuration manager, the payload data is initially written into dedicated asynchronous FIFO buffers. The recipient module constantly polls its dedicated read buffer to read data from it, if any. Using asynchronous buffers allows for error-free inter-clock domain communication. The router drops the packet if no match is found in the address table for a packet's destination address. Lastly, any new communication session is recorded in the ''session memory'' so that only one user can establish a secure connection with an accelerator at any given time. This specifies the way data is received into an FPGA.

As for data transmission from the FPGA to the external network, data from other modules are written into asynchronous buffers. The network manager reads from these buffers and encapsulates the payload with necessary header information into a packet for communicating via TCP/IP protocol. For example, if the payload contains accelerator results generated in the FPGA, then the destination address in the IP header of the packet is the one from the user who established the session with the accelerator. Once the packet is ready, the physical transmitter sends it out via the appropriate port.

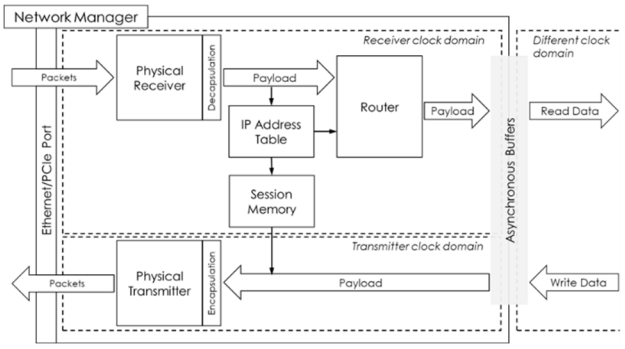

FIGURE 2. Architecture overview of the network manager.

D. CONFIGURATION MANAGER

The configuration manager is a static hardware module mainly responsible for creating the user's accelerator design in the FPGA. Accelerator designs come in the form of bitstreams or bit files. Application hardware logic is sent as partial bitstreams over the network from a host machine using a toolchain provided by the FPGA vendor. The network manager receives bitstreams within the payload content of Ethernet packets, decapsulates the packets, extracts the bitstreams, and writes to buffers from which the configuration manager can read the bit files. Moreover, the configuration manager uses volatile memory to hold bitstreams and a dedicated controller submodule to communicate with the ICAP interface [9], [33]. DPR uses the ICAP controller to download bitstream data into dynamic regions (PRRs) and reconfigures the regions to create the accelerator hardware specified in the partial bitstreams. Once an accelerator is configured, the configuration manager informs the vFPGA manager of the new accelerator's physical location. The network manager receives an acknowledgment that an accelerator has been successfully created. Next, the network manager assigns the accelerator a unique IP address, and the address table stores this information. Users can then establish TCP sessions with their respective accelerators to send and receive data.

E. ADAPTER INTERFACE

Accelerators in the FPGA require a static interface to exchange data with static logic modules such as the network manager. Irrespective of the design of an accelerator, the approach in which it will communicate with the network manager does not change. Hence, the communication interface must be static. According to [9], the interface is called a wrapper. However, in this work, we refer to it as the adapter interface or an adapter. As shown in Fig. 3, an adapter consists of buffer memories, serializer, deserializer, bit packer, and bit unpacker submodules. It has a read and a write buffers where the accelerator can read-from and write-to, respectively. Data are moved in chunks of bits called (data) words. Submodules inside the adapter are used to change the size of data word between modules. This is mainly to resolve any mismatch in word length when static logic modules send data in specific

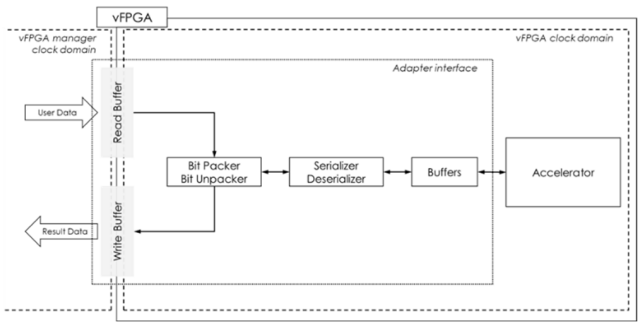

FIGURE 3. Architecture overview of adapter interface.

sized words and user accelerators read words in a different size, and vice-versa. Moreover, when the user's accelerator design code is converted into partial bitstreams, the adapter module is automatically generated, converted into a partial bitstream, and included within the same files. This way, the user is not required to write the logic for the adapter. However, the user must preset specific parameters of the adapter interface, such as the number of input and output channels and the width (i.e., the number of bits or size of the word allowed) of each channel.

F. VIRTUAL FPGA (vFPGA)

An accelerator may require more than one homogenous dynamic region, especially if it requires resources that cannot be fulfilled by a single region alone. However, the number of resources required for a hardware design cannot be determined ahead of bitstream reconfiguration. This means that once the configuration manager configures the accelerator in the FPGA fabric, we can only identify the number of resources utilized. Therefore, we only configure a set of predefined accelerators whose required number of CLBs are already known and partial bitstreams generated. This is typically done by cloud service providers that offer AaaS. When the user selects the desired accelerator from a list of options, corresponding partial bitstreams for that accelerator are transmitted to an available FPGA in the network for reconfiguration. A predefined schematic of each accelerator is stored as a netlist in a database. Upon selecting an accelerator, the corresponding netlist and the adapter interface logic are converted into partial bitstreams.

An accelerator requires an adapter interface to communicate with the static logic modules and may require one or more PRRs for configuration. We encapsulate the accelerator and its adapter module inside a single entity called virtual FPGA (vFPGA). From the user's perspective, they get the illusion that their accelerator is operated by a single, dedicated, physical FPGA. However, that is not the case physically. Each accelerator runs by one vFPGA only, and there is secure logical isolation between different vFPGAs. Fig. 4 shows three vFPGAs configured in a physical FPGA where each vFPGA contains an accelerator hardware and an adapter interface. A vFPGA can be configured across one or more PRRs and across more than a single physical FPGA.

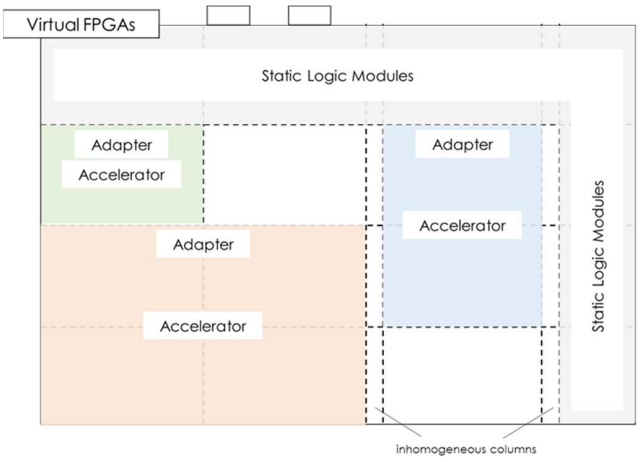

FIGURE 4. Encapsulation of accelerator & adapter modules into vFPGA.

This idea of encapsulating an accelerator and an adapter module to create a vFPGA is different from the work done in [9], in which a vFPGA represents a PRR and might be either occupied with an accelerator hardware or empty.

G. vFPGA MANAGER

Each FPGA can contain one or more vFPGAs depending on the number of resources the FPGA can provide and the number of resources each vFPGA demands. Therefore, this requires a vFPGA manager, as shown in Fig. 5. The vFPGA manager is a module that is responsible for monitoring and maintaining vFPGAs. Each vFPGA, upon instantiation, is given a unique ID by the vFPGA manager for internal addressing. A vFPGA can be in one of the two states – idle or busy. It is idle if its accelerator is not currently processing a task and is otherwise busy. The manager constantly monitors the status of each vFPGA instance, whether idle or busy. Moreover, since each accelerator must be addressable to enable communication with the user, assigning and retracting IP addresses concerning vFPGAs are performed by the vFPGA manager. This is done by maintaining two tables; one table contains IP address-to-vFPGA ID mappings, and the other contains vFPGA ID-to-physical address of the vFPGA. Whenever the network manager routes data to this manager, it looks up the destination IP address in the first table to find the corresponding vFPGA ID. Then, it looks up the second table to find the physical address of the vFPGA. Therefore, the vFPGA manager forwards the application data from the network manager to the intended accelerator. This is the core element of the implemented virtualization framework in this work, where physical address spaces are mapped to virtual address spaces.

Similarly, when an accelerator attempts to send back data to its user, the manager looks for the physical address of the network manager and writes the data into appropriate buffers for the network manager to read. In addition, all communication between the accelerator inside vFPGA and the vFPGA manager happens via the adapter interface that was mentioned

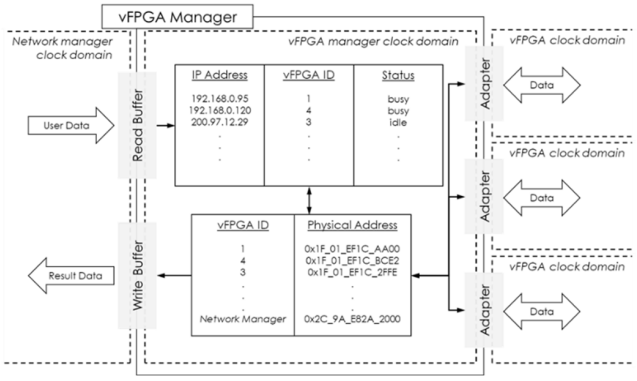

FIGURE 5. Architecture overview of the vFPGA manager.

earlier. The read and write buffers in the adapter allow the vFPGA to read from and write to, respectively. For a vFPGA to read user input data, the manager writes the data into the read buffer of the adapter. For a vFPGA to send result data, it writes into the write buffer of the adapter so that the vFPGA manager can read and route the data further to the network manager.

H. FPGA HYPERVISOR

In Fig. 1, the vFPGA manager and the configuration manager are collectively labeled as ''hypervisor''. Its role is discussed in the literature as the software that enables virtualization. The hypervisor has frontend and backend functions. The key difference in our framework compared to other frameworks in the literature is that the hypervisor is split into two separate modules – the vFPGA manager and the configuration manager. The hypervisor backend includes the configuration manager and parts of the vFPGA manager. Thus, the backend functions include initializing and partitioning the FPGA, configuring vFPGAs from partial bitstreams, and routing user data to and from vFPGAs.

On the other hand, frontend functions include processing requests for new accelerator creation, getting the status of created vFPGAs, initiating data transfers to and from user's accelerators, and requesting termination of accelerators. The vFPGA manager provides all these functions in the framework. Moreover, there is a distinction in the implementation between the hypervisor frontend and the backend. The backend is usually coded in Verilog and resides on the FPGA fabric, whereas the frontend resides in a host machine with network capabilities to reach the FPGA. Both front and back ends are connected via a TCP stream. Thus, when a user from the host machine sends any command to the hypervisor frontend, it triggers the corresponding function in the backend and processes the user requested function. In this approach, FPGA hypervisors establish a practical and error-free communication between FPGAs and other cloud entities such as host machines and are essential to the virtualization framework.

I. FPGA RESOURCE POOL

Further to the hypervisor capabilities, this section discusses the ability to pool FPGA resources in the cloud infrastructure. The result of virtualization is to obtain a pool of abstract resources that can be quickly and efficiently provisioned to tasks. To achieve this, one of the backend functions of the FPGA hypervisor is to track the number of occupied and unoccupied regions at any given time. With this, we can obtain the total number of unutilized regions across all FPGAs in the data center at any given time. Therefore, we can generate a pool of available FPGA regions using the hypervisor of each FPGA that monitors and tracks the number of regions. Upon provisioning any FPGA region for configuring an accelerator, the hypervisor of that FPGA will notify a centralized control module, i.e., a unified manager, that the total number of resources in the pool has been reduced by one. Thus, the resource pool is constantly updated owing to the communication between FPGA hypervisors and the control module.

IV. SCHEDULING PROBLEM MODEL

As a result of implementing the proposed virtualization framework, we obtained a pool of FPGA resources from which resources can be provisioned to cloud tasks. In this context, a cloud task refers to a user request for configuring an accelerator, whereas a resource refers to a PRR in an FPGA chip that resulted from the partitioning process. The FPGA resource allocation must effectively manage the cloud resources and execute the consumers' tasks. Hence, this section presents an integer linear programming (ILP) optimization model for the FPGA to minimize the competition time of tasks within the pool of FPGA resources. The work assumes a cloud datacenter infrastructure with computing servers, limited FPGA-based accelerators, and networking resources that interconnect compute-to-compute and compute-to-FPGA resources. Moreover, the model assumes the following in the cloud computing environment:

- FPGA fabrics are homogeneous with regard to their architecture, and therefore, the model considers that all FPGA devices within the cloud infrastructure are from the same vendor family. The validity of the assumption holds since the architecture of the configurable logic block (CLB) in the fabric of an FPGA differs from vendor to vendor. Cloud providers such as Amazon Web Services (AWS) and Microsoft Azure provide their consumers with accelerator services by using only one FPGA family within their datacenters [34], [35], [36], [37].
- Resource blocks or PRRs obtained within a single FPGA, because of the partitioning process, are all identical.
- The number of regions needed for each accelerator is a priori knowledge. Every accelerator request from the users' tasks has a specific number of required regions.
- The execution time of each acceleration task is a known parameter measured in time units. This is true in an AaaS-enabled cloud infrastructure.
- The deadline of each task is a known parameter measured in time units. Assigning a deadline is crucial in the

model to prioritize tasks. For instance, tasks with shorter deadlines have higher priority.
- Tasks are independent. This assumption applies only in scenarios where acceleration tasks are considered independent of each other and an acceleration task's dependencies, which may exist within its subtask level, are ignored [13], [38].
- No preemption of tasks. Once resource blocks are allocated to a task, the task execution must finish without interruption.

A. OPTIMIZATION MODEL

As discussed in section III, a pool of FPGA resource blocks results from the partitioning process. Every FPGA is partitioned as soon as an FPGA instance is initialized. Moreover, a task is a request to create an accelerator using the FPGA resource pool when a consumer selects an acceleration service. This section formulates the minimization of the completion time of all tasks as they are allocated to a pool of FPGA resources. The objective function Z, presented in Equation 1 of Model 1, minimizes the makespan (the max completion time). It is a min-max function where it first finds the maximum end time of tasks. The end time of task k is calculated by adding the start time (t_k^S) and execution time (t_k^E). The subtraction by 1 in the equation accounts for the fact that the simulation begins from t = 1, giving the correct end time value. Moreover, since the last task to finish execution will always yield the highest end time value, the objective function thus considers the end time of the last task as the maximum. The function minimizes the maximum, and as a result, it minimizes the makespan. The objective function is formulated as shown in Model 1.

$$Z = \min \max_{k=1,\dots,K} (t_k^S + t_k^E - 1) \quad (1)$$

s.t.

$$\sum_{k=1}^{K} x_{b,k,t} \le 1, \ \forall b \in \{1,\dots,B\}, \forall t \in \{1,\dots,t^{max}\} \quad (2)$$

$$t_k^S + t_k^E - 1 \le t_k^D, \quad \forall k \in \{1,\dots,K\} \quad (3)$$

$$\sum_{b=1}^{B} x_{b,k,t} = C_k, \ \forall k \in \{1,\dots,K\}, t = t_k^S \quad (4)$$

$$x_{b,k,t'} = x_{b,k,t''}, \quad \forall b \in \{1,\dots,B\}, \quad \forall k \in \{1,\dots,K\}, t' = t_k^S, \ t'' = t_k^S + t_k^E - 1 \quad (5)$$

$$\frac{\sum_{t=1}^{t^{max}} \sum_{b=1}^{B} x_{b,k,t}}{t_k^E} = C_k, \quad \forall k \in \{1,\dots,K\} \quad (6)$$

$$x_{b,k,t'} - \sum_{t=1}^{t^{max}-1} (sign(x_{b,k,t} - x_{b,k,t+1} + 1) - 1) \le 1, \quad \forall b \in \{1,\dots,B\}, \quad \forall k \in \{1,\dots,K\}, t' = 1 \quad (7)$$

$$x_{b,k,t} \in \{0, 1\}, \ \forall b \in \{1,\dots,B\}, \quad \forall k \in \{1,\dots,K\}, \forall t \in \{1,\dots,t^{max}\} \quad (8)$$

$$t_k^S \in \{1,\dots,t^{max}\}, \ \forall k \in \{1,\dots,K\} \quad (9)$$

MODEL 1. **ILP optimization model.**

Constraint (2) ensures that a block can be part of only one task at a time. That is, a resource block may be allocated to at most one task at any point in time. Two or more tasks cannot share the same resource block simultaneously.

Constraint (3) ensures that any task's execution must be completed before its deadline. This is performed by checking whether a task's start time and execution time are less than or equal to its deadline.

Constraint (4) ensures a task must have all the required blocks ready at its start time. Therefore, it ensures that the number of blocks allocated at a task's start time equals the number of blocks required.

Constraint (5) ensures that a task must have the same block allocation at its start and end times. Blocks allocated at the start time should remain allocated until the end time of a task.

Constraint (6) ensures that a task can be allocated to a specific number of blocks for a fixed number of time units. While constraint (4) checks for the correct number of blocks allocated at the start time, constraint (6) ensures that the same number of blocks remain consistently allocated from the start to the end time of a task.

Constraint (7) ensures that blocks will continue executing the task until it is completed without any interruption. Hence, this Constraint safeguards the no-preemption assumption.

Constraint (8) is a binary variable that indicates whether a task is allocated to a block at a point in time.

Constraint (9) limits the start time of a task to be between 1 and the maximum deadline of tasks t^{max}.

Table 1 summarizes the notations and their respective descriptions.

TABLE 1. **Summary of notations.**

Notation	Description
b	b^{th} block resource in the FPGA pool. A single FPGA resource can be referred as a resource block, partition, slot, partially reconfiguration region (PRR), or dynamic region
B	total number of blocks in the FPGA pool
k	k^{th} task
K	total number of tasks
t	t^{th} time unit
t_k^S	start time of task k
t_k^E	execution time of task k
t_k^D	deadline of task k
t^{max}	maximum deadline of tasks
$x_{b,k,t}$	0, if block b is not allocated to a task, k at time unit t 1, if block b is allocated to a task, k at time unit t
C_k	number of required blocks by task k

B. MODEL VALIDATION

The proposed model is validated for small-scale problem instances using IBM ILOG CPLEX Optimization engine that implements the branch-and-cut exact solution method. CPLEX ran on a Windows machine with 16 GB DDR4 DRAM at 3000 MHz and a 6-core processor at 3.6 GHz. Four experiments with an increasing number of tasks and resources

TABLE 2. Experiment 1 Task specifications.

Task ID	Execution Time	Blocks Required	Deadline	Blocks in Pool
1	2	2	2	2
2	1	1	4	2
3	1	1	5	2

TABLE 3. Experiment 2 Task specifications.

Task ID	Execution Time	Blocks Required	Deadline	Blocks in Pool
1	2	2	2	5
2	1	1	4	5
3	1	1	5	5
4	3	4	11	5
5	3	3	6	5
6	3	4	8	5

TABLE 4. Experiment 3 Task specifications.

Task ID	Execution Time	Blocks Required	Deadline	Blocks in Pool
1	2	2	3	5
2	4	1	6	5
3	1	3	8	5
4	1	2	8	5
5	2	5	11	5
6	5	1	12	5
7	5	2	13	5
8	2	3	14	5
9	2	4	18	5
10	4	1	18	5
11	3	2	20	5
12	3	2	20	5
13	4	3	24	5
14	3	3	27	5
15	1	1	30	5

TABLE 5. Experiments with varying tasks and resources.

Experiment	Tasks	Resource Blocks
1	3	2
2	6	5
3	15	5
4	1000	100

are conducted, and an exact solution for each problem is sought. Tables 2, 3, and 4 show the different number of tasks and resources for the first three experiments. A task is defined by execution time, required resource blocks, and deadline. The resource blocks refer to PRRs that are obtained because of logically partitioning an FPGA. A summary of all four experiments can be found in Table 5.

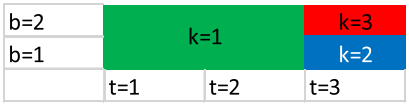

FIGURE 6. Experiment 1 final solution using exact method.

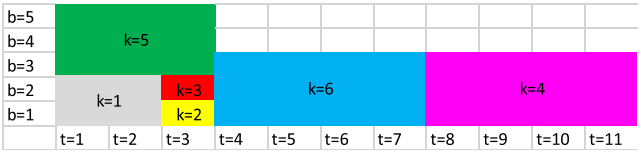

FIGURE 7. Experiment 2 final solution using exact method.

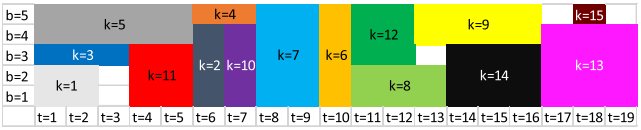

FIGURE 8. Experiment 3 final solution using exact method.

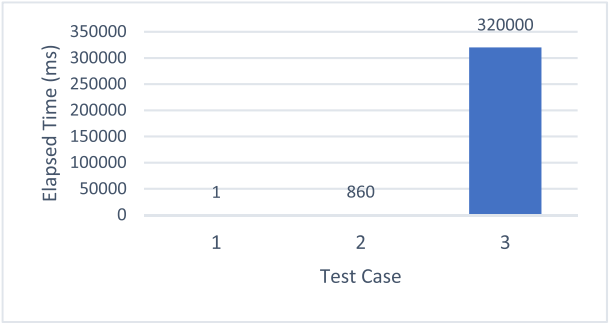

FIGURE 9. Elapsed time for scheduling using exact method.

The created model was validated with the exact solution method for the first three experiments and achieved the results presented in Fig. 6, Fig. 7, and Fig. 8, respectively. The figures show the discreet time units t, the FPGA blocks as b, and the allocated task k into a block at a specific time slot.

Fig. 9 illustrates the elapsed time (y-axis) on CPLEX for finding the exact solution in each experiment (x-axis). The simulation times of these experiments are ∼1 ms, 860 ms, and 320,000 ms, and the makespans are 3, 11, and 19, respectively. Although the reported solution for the third experiment was after running CPLEX for around 5 minutes, we allowed CPLEX to run for over 20 hours to find the optimal solution. The entire search space could not be exhausted, and an improved solution was not found. For the fourth experiment, CPLEX could not handle the dataset size.

In the exact solution approach used by CPLEX, the number of permutations grows exponentially with an increase in the number of tasks and resources, as shown in Fig. 9. Exploring the entire search space is infeasible due to the exponential growth in time complexity. The presented scheduling problem is NP-hard and requires a heuristic-based approach that ensures a near-optimal solution. The following section introduces the proposed heuristic approach.

V. PROPOSED HEURISTIC SOLUTION

Given the modeled scheduling problem in section IV, this section provides a detailed description of the implemented heuristic algorithm. The inspiration for the algorithm comes from the process of annealing. In this technique, the analogy of a material is heated to a melting point and then cooled in a controlled environment to increase or decrease the size of its crystals. Simulations of such behavior are used to solve optimization problems. The initial temperature and the cooling rate are essential parameters in simulated annealing (SA).

SA accepts worse neighboring states based on an acceptance probability function. This function makes the SA a metaheuristic and enables it to overcome the local optima problem in heuristic-based approaches. Moreover, the acceptance probability function considers the following variables – current state energy, neighboring state energy, and temperature. If the neighboring state energy appears to be worse than the current state energy, then given the current temperature, either accept or decline the move. Usually, higher temperature values produce greater probabilities of accepting a worse neighbor. As the temperature reaches zero, the SA declines to accept worse neighbors and becomes more inclined to move with better neighbors. On the one hand, this indicates that the cooling rate controls how fast the temperature reaches zero and stops accepting worse states. On the other hand, the initial temperature is responsible for determining how much worse off the energy of a neighboring state can be before the SA declines the move. Lastly, a very high initial temperature with a very low cooling rate would explore more of the search space, almost guaranteeing the optimal solution but at the cost of execution time. The system's behavior at every fixed temperature in the cooling profile can be investigated using the Metropolis algorithm – a significant component of the SA.

A. SA IMPLEMENTATION

This section discusses our implementation of the customized SA algorithm fused with the Metropolis algorithm to minimize the makespan specified in section IV as depicted by the pseudocode shown in Algorithm 1. The initial solution sorts out the list of tasks in the increasing order of their deadlines. The order of the tasks is the current state in the SA, and resources are immediately allocated to the tasks in their respective order. Makespan of these tasks is the energy of the current state. In the next iteration of the Metropolis algorithm, we randomly select two tasks and swap their positions in the list. This step is known as perturbation. Perturbation is defined as a modification to the state of a system, and this modification occurs from a source external to the system. In our algorithm, the perturbation is a random swap of two tasks in the dataset that is modifying the current state. This changes the original order of the tasks and thus, creates a new order or a neighboring state. Once all tasks have been scheduled in the new order, the makespan becomes the energy of the neighboring state. Based on the acceptance probability function and the initial temperature, the perturbation is either accepted or rejected, and accordingly, the neighbor becomes the current state and the starting point of the next perturbation. This process continues iteratively, and the temperature reduces in each iteration based on the predefined cooling rate. The process terminates if the SA does not find a better neighbor for a certain number of iterations or the temperature reaches close to zero. Reaching the predefined iteration threshold is known as the convergence of the solution. We have chosen 16 to be the iteration threshold based on several parameter-tuning experiments.

Algorithm 1 Implementation of Simulated Annealing

Input: Task list that consists of task execution time, required blocks and deadline;
Initial configuration: Task list is sorted using earliest deadline first X_{soln};
Determine initial temperature T(0);
Determine freezing temperature T_f;
while (T(i) $>$ T_f and not converged) ***do***
repeat
  Perturb (X_{soln}) by swapping two tasks randomly;
  Find neighbor solution X_{new};
  Compute $\Delta Z = \text{cost}\,(X_{now} - X_{soln})$;
  if ($\Delta Z \leq 0$) ***then***
    Update X_{soln}; /*accept perturbation*/
  else if (random (0, 1) $< e^{-\Delta^{Z/T(i)}}$) ***then***
    Update X_{soln};
  else
    Reject X_{new};
  endif
  endif
until thermal equilibrium
Save best-so-far X_{soln};
Check convergence
$T(i+1) = \alpha T(i)$; /* cooling schedule */
endwhile

B. TASK LIST GENERATION

In section IV, the model was validated using a very small number of tasks and resources, given the limitation of the exact solution method in CPLEX. Besides the gain in performance in finding a solution in comparison with an exact method, heuristics solutions must also be scalable to handle large problem sizes. To test the scalability of the proposed algorithm, we generated a long list of tasks with an adequate pool of resources such that there exists a feasible solution.

Recall that each task needs to have a specific number of required FPGA resource blocks or PRRs, execution time, and deadline. We use Poisson distribution to randomly retrieve a value for both the number of blocks and the execution time. The Poisson distribution is a discrete random distribution that gives the probability of several events occurring over a fixed time interval. It assumes that events occur at a constant rate and each event occurs independently of the time since the last event. In contrast to a continuous normal distribution, the

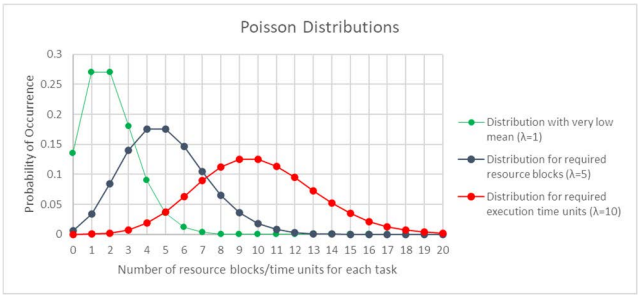

FIGURE 10. Various poisson distributions used in task generation.

Poisson distribution is based on discrete values and is more applicable when dealing with integer counts. It has numerous practical applications, such as a random number of tasks arriving at a data center and the random delay between every two tasks [39]. Furthermore, the main characteristic of this distribution is the mean indicated by lambda (λ) in Fig. 10. A different value of mean signifies a different shape of the distribution, and as we choose a higher mean, the distribution becomes closer to a normal distribution. Real-world scenarios tend to follow a Poisson distribution [39]. Therefore, we use various Poisson distributions to determine two random variables during task generation: the required number of resource blocks and execution time in a discrete-time unit. For the number of resource blocks, the mean distribution is kept at $\lambda = 5$, and for execution time units, the mean is kept at $\lambda = 10$. To comprehend and apply the graphs in Fig. 10, let us suppose that we are generating a new list of tasks to specify the required resources for each task. The probability that a task may require exactly 5 blocks is 0.175. On the other hand, the probability that the required execution time would be a value between 1 and 20 is $\sim$1. Note that the first distribution with a very low mean ($\lambda = 1$) as no use as such but we included to illustrate how the selection of mean can produce a different distribution.

From this, the initial tendency is to think in the lines of a 3D Boolean array, where the dimensions are task number, block number, and time unit, and any cell in the array is either 1 to indicate occupied or 0 for empty. However, other possible representations still exist, and we aim to represent the solution in a way that incurs the minimum amount of constraint violations possible. Constraint violations may occur at the time of swapping two tasks randomly in our algorithm and when scheduling them to resources.

We propose a 1D representation of the 3D array mentioned earlier such that the array cells will hold the unique ID of a task based on whether the task is allocated to block b at time t. The index value i of the array cell in which this task ID exists gives information on the block number and the time unit. Fig. 11 shows a conceptual diagram of the solution representation (X_{soln}), where b is the block number, t is the time unit, and every cell in the array holds the ID of the task allocated to block b at time t. The array index i starts at 0, and the total number of blocks is denoted by B, whereas total time units are denoted by t^{max}.

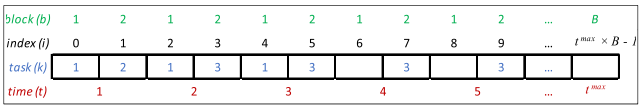

FIGURE 11. Solution representation using 1D array.

Xsoln undergoes several perturbations during the annealing process until it reaches an equilibrium where the final task allocation represents the best solution. The initial configuration is obtained by allocating the tasks using the earliest deadline first algorithm, where the task with the earliest deadline is scheduled first, followed by the next task until all tasks have been allocated.

The 1D array representation is by far the best alternative because the only Constraint that must be checked during the perturbations is the deadline constraint. This contrasts with a 3D Boolean array which was the initial alternative and too many constraints must be validated to determine the feasibility of the solution. We also devised a linked list representation where each time node is connected to the next time node in a singly linked list. Each time node is also connected to another singly linked list which consists of resource blocks or resource nodes.

C. METROPOLIS STEP AND FEASIBILITY

There are three main components to the Metropolis step, namely, the perturbation, the acceptance criteria, and the thermal equilibrium criteria. We start by perturbing the existing solution X_{soln} by randomly selecting two tasks in the task list and swapping the order in which they appear on the task list. After that, we attempt to schedule the task list by taking them in the order of task deadline yielding X_{new}. While doing so, only the deadline constraint, i.e., Constraint (4), needs to be rechecked to ensure that deadlines are not violated. Next, the acceptance criterion outlined in Algorithm 1 checks the change in the objective function, $\Delta Z = Z(X_{new}) - Z(X_{soln})$. If the change due to perturbation reduces the objective function, the perturbation is accepted and X_{soln} becomes X_{new}. In other words, if the makespan of the X_{new} schedule is smaller than that of the X_{soln} schedule, which is the best-so-far, X_{soln} is updated to X_{new}. On the other hand, if the perturbation causes an increase in the objective function, it will only be accepted with a probability of $e^{-\Delta Z/T(i)}$. The acceptance criterion applies only to perturbations yielding a feasible X_{soln}. Furthermore, the inner loop in the algorithm deals with thermal equilibrium. As more neighboring solutions are found for the same temperature value, it is said that the algorithm is reaching thermal equilibrium. Hence, thermal equilibrium is nothing more than a predefined number of iterations for the inner loop. We set the thermal equilibrium criterion to be one-third of the dataset size.

D. COOLING SCHEDULE

The initial temperature *T(0)* yields a high acceptance probability of around 0.8 for moving to worse neighboring states. On the other hand, the freezing temperature yields a very

small acceptance probability of around 2^{-25}, rendering worse neighboring moves impossible, and hence only better neighboring states are allowed. The cooling schedule used in our work is $T(i+1) = \alpha T(i)$, where $\alpha = 0.9$. The symbol α denotes the cooling rate of the temperature for the next iteration.

E. CONVERGENCE

While searching the space for a task order that potentially has a lower makespan, the SA algorithm saves the best-so-far solution that yields the smallest Z. This ensures that the returned solution is the best obtained regardless of the terminating temperature of the SA algorithm. Convergence is then achieved when the best-so-far solution does not change for several iterations. Once 16 iterations have passed with no change to the solution, the process stops and considers best-so-far as the final solution.

F. HEURISTIC COMPLEXITY ANALYSIS

The initial configuration of the algorithm is obtained by sorting the tasks based on the shortest deadline first. This has a computational cost *Klog(K),* where K is the total number of tasks. Once tasks are sorted, we obtain the initial solution considering all but Constraint (3) and allocating each task to a specific number of FPGA blocks at a specific time unit. Thus, the complexity of the allocation process is given by KBt^{max} where B is the total number of blocks in the resource pool, and t^{max} is the maximum task deadline. Furthermore, in Algorithm 1, it can be observed that the outer *while* loop depends on two conditions, the current temperature value and whether convergence is achieved. Our solution defines convergence to be achieved when the best-so-far solution does not change for 16 outer loop iterations. The iteration count is defined through parameter tuning experiments.

This entails that the complexity can be denoted by M as the number of iterations to occur until convergence is found. Next, the inner loop repeats the block until thermal equilibrium is achieved. MK/3 gives the computational cost involved because the inner loop runs as many times as one-third of the task size. Then, within this loop, several steps are carried out of which one is the feasibility check on the perturbed X_{soln}. Only the deadline constraint is rechecked to ensure a feasible perturbed Xsoln. The computational cost, therefore, as evident from Constraint (3) in Model 1, is some constant denoted by C. Since we allocate once at the beginning after task sort and then reallocate in the inner loop for each perturbation, the algorithm complexity can be written as shown in Equation 10. Equation 11 shows a further simplified version of the previous equation. It can be noted that increasing the total number of tasks (K) will have a greater impact on the performance of the algorithm than increasing total resource blocks (B) or maximum deadline (t^{max}).

$$K\,log\,(K) + KBt^{max} + \frac{MK}{3} \times KBt_{max} \times C \qquad (10)$$

$$K\,log\,(K) + KBt^{max}(1 + \frac{MKC}{3}) \qquad (11)$$

TABLE 6. Task specifications of datasets for parameter tuning.

Size	Number of Tasks	Mean Execution Time	Mean Number of Blocks
Small	10	5	10
Medium	50	10	50
Large	200	15	100

G. PARAMETER TUNING

Algorithmic parameters can affect the simulation time heavily. In the proposed SA, the simulation terminates when the solution does not change for a set number of iterations. Therefore, the iteration threshold becomes the algorithm's termination condition, and we must tune this parameter to reduce the simulation time as much as possible. At the same time, minimizing the simulation time must not compromise the quality of the solution too much.

To conduct the parameter tuning experiments, we consider 3 datasets of different sizes in terms of the number of tasks, the execution time for each task, and the number of blocks each task requires. The problems were inputted through the proposed algorithm, varying the maximum number of iterations allowed from 2 to 128 as 2, 4, 8, 16, 32, 64, 128. The solution behavior in terms of improvement in the objective function and the degradation in the incurred simulation time trying to converge. Table 6 shows the specifications of different dataset sizes. Note that through all parameter tuning experiments, we keep the resource pool constant with 1000 FPGA blocks, ensuring sufficient resources for all tasks in each dataset.

Fig. 12 and Fig. 13 show the objective value of the proposed schedule and the simulation duration respectively for the small dataset size against number of iterations. It is apparent from Fig. 12 that the algorithm was able to obtain the best solution of 6-time units within the first 2 iterations. Clearly, increasing the number of iterations does not help reduce the objective any further as it seems that 6-time units is the minimum makespan for the tasks. Furthermore, Fig. 13 shows that increasing the number of iterations degrades performance, since the elapsed time of the simulation for a greater number of iterations grows larger. Therefore, we conclude that for a small problem, 2 iterations are sufficient to obtain a suboptimal schedule. Running the algorithm for 2 iterations on the small dataset takes an average of 1 millisecond.

Fig, 14 and Fig. 15 show the objective of the proposed schedule and the time for which the simulation elapsed, respectively, for the medium dataset plotted against the number of iterations. Fig. 14 shows that as the number of iterations increases, a better scheduling solution with a shorter makespan is produced. This is because we allow the algorithm to run for a larger number of iterations before termination, increasing the possibility of finding a better schedule. Moreover, Fig. 15 further confirms our previous observation that an increase in the number of iterations increases simulation duration. We conclude from the two experiments that a

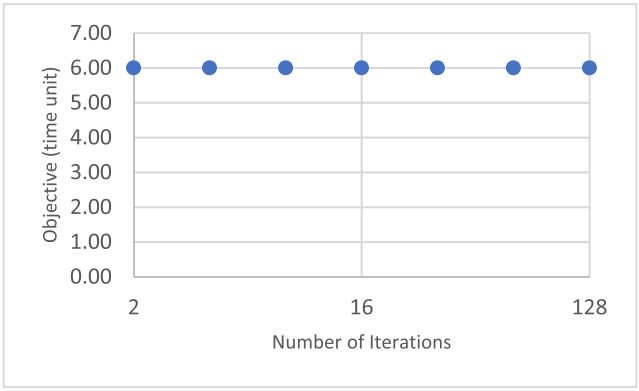

FIGURE 12. Objective vs. the number of iterations for a small dataset.

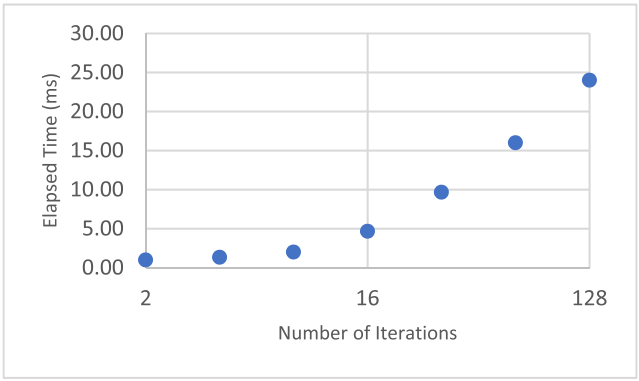

FIGURE 13. Elapsed time vs. the number of iterations for a small dataset.

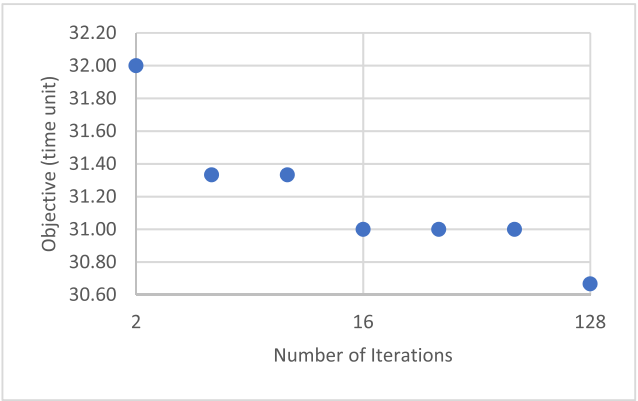

FIGURE 14. Objective vs. the number of iterations for a medium dataset.

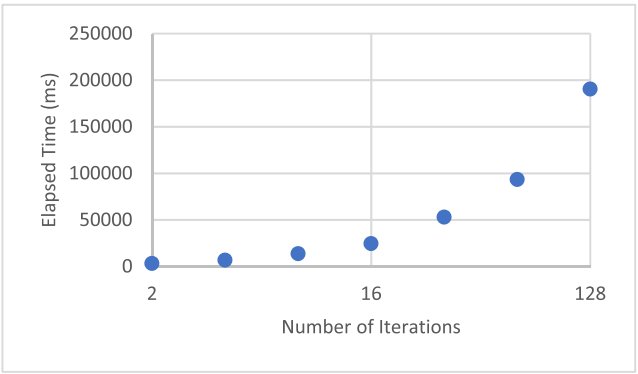

FIGURE 15. Elapsed time vs. the number of iterations for a medium dataset.

reasonable schedule is obtained with an objective of around 31-time units using 16 iterations. The choice of 16 iterations is because it provides a near-optimal solution while avoiding longer simulations incurred by higher iteration counts. Hence, 16 iterations on the medium dataset would take an average of 690 milliseconds.

Fig. 16 and Fig. 17 show the objective of the proposed schedule and the simulation duration respectively for the large dataset against the number of iterations. Similar to the medium dataset experiment, Fig. 16 shows that as the number of iterations increase, we obtain a better schedule with a shorter makespan. Hence, we conclude from the experiment that we achieve a reasonable schedule with a makespan of around 152-time units using 16 iterations. The 16 iterations seem to be adequate for finding a suboptimal solution. On the large dataset, the simulation takes on average 24000 milliseconds (24 seconds).

H. ADAPTIVE SIMULATED ANNEALING

The previous sections discussed the various aspects of the proposed SA algorithm with static scheduling. With static scheduling, a batch of tasks arrive at the data center, and the SA schedules them to the available resources. However, cloud computing is a dynamic environment. Hence, the SA must be adaptive to support newer tasks dynamically arriving at the task queue and obtain a schedule including both previously unscheduled and newly queued tasks. Hence, Algorithm 2 shows the adaptive SA that differs from the SA in steps A-1 to A-6. First, the algorithm deals with the initial batch of tasks in the task list and evaluates as per the proposed model's objective function (see Equation 1). After convergence is achieved, it saves the best-so-far schedule and calls a delay function until a new batch of tasks arrives. The delay function is based on a random value from a Poisson distribution. The task list is updated with newly arrived tasks in ascending order of task deadlines. Then, it starts over the process of finding a new schedule with a minimum makespan that considers tasks from both the previous batch and the new batch. However, previous tasks whose execution had already started are excluded from the rescheduling process (i.e., no preemption of tasks) and only the tasks that did not start being executed are passed forth. In this way, the adaptive SA enables dynamic scheduling of tasks which is commonly used in cloud environments.

VI. EXPERIMENTATION AND RESULTS

This section presents the conducted experiments and the achieved results to validate the proposed virtualization framework and heuristic algorithm. Multiple experiments were conducted to evaluate the exact solution and the proposed heuristic solution. The quality and performance of the solutions yielded by both methods are compared. We further

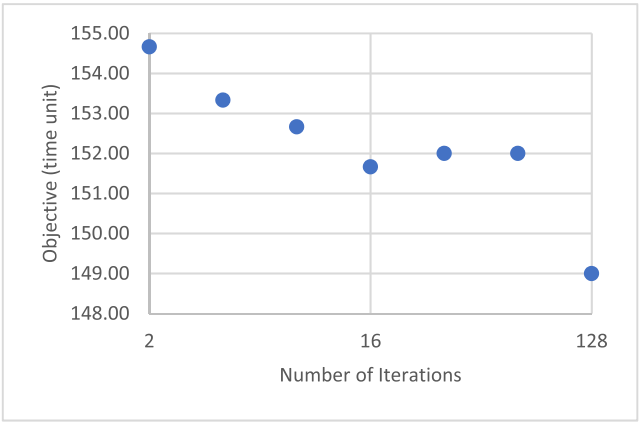

FIGURE 16. Objective vs. number of iterations for large dataset.

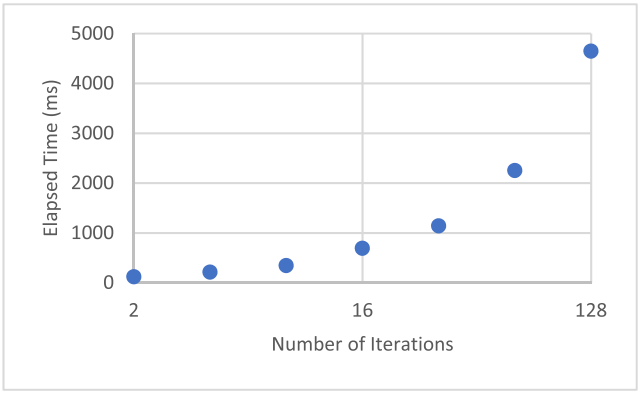

FIGURE 17. Elapsed time vs. number of iterations large dataset.

compare the performance of the proposed heuristic solution with two classical allocation algorithms – First Come First Serve (FCFS) and Shortest Deadline First (SDF). Those algorithms are commonly used in cloud data centers.

A. SIMULATION ENVIRONMENT FOR SA SCHEDULER

To experiment with the proposed heuristic method, we use the same physical environment as the one used to validate the exact method, i.e., a Windows machine equipped with 16 GB DDR4 DRAM at 3000 MHz and a 6-core processor at 3.6 GHz. Moreover, the SA is implemented using Java, whereas the exact method is implemented using the Optimization Programming Language (OPL) and the CPLEX optimization engine. The implementation of the SA algorithm is a standalone module integrated with the CloudSim toolkit and validated in the next section B. The results show that the proposed heuristic algorithm outperforms in comparison to two known techniques: first come, first served (FCFS) and shortest deadline first (SDF).

B. VALIDATION OF THE SA SCHEDULER

The three experiments that were used to validate the exact solution method were applied to the SA algorithm. Table 2, 3, and 4 provide task specifications of the first three experiments. As discussed in section V, the SA first finds

Algorithm 2 Implementation of Adaptive SA

Input: Task list that consists of task execution time, required blocks and deadline;
Initial configuration: Task list is sorted using earliest deadline first X_{soln};
Determine initial temperature T(0);
Determine freezing temperature T_f;
A-1: Current schedule is empty;
A-2: ***repeat***
while (T(i) $>$ T_f and not converged) ***do***
repeat
Perturb (X_{soln}) by swapping two tasks randomly;
Find neighbor solution X_{new};
Compute $\Delta Z = \text{cost}\,(X_{new} - X_{soln})$;
if $(\Delta Z \leq 0)$ ***then***
Update X_{soln}; /*accept perturbation*/
else if $\left(\text{random}(0, 1) < e^{-\Delta Z(i)}\right)$ ***then***
Update X_{soln}
else
Reject X_{new}
endif
until thermal equilibrium
Save best-so-far X_{soln};
Check convergence;
$T(i + 1) = \alpha T(i)$; /* cooling schedule */
Endwhile
A-3: Current schedule = best-so-far X_{soln};
A-4: Delay (***based on Poisson distribution***);
A-5: Update task list with new set of tasks;
A-6: ***until*** (true)

an initial solution that might be infeasible, and then it iteratively converges to suboptimal feasible solutions. Fig. 18 and Fig. 19 show the initial and the final solutions, respectively, that were obtained in less than a millisecond of the simulation time. In these figures, b represents a single FPGA resource block or PRR, k is a task, and t is the unit time. As the heuristic approach finds a schedule for the tasks, resources are allocated to each task for a specific duration of time. Moreover, both the exact and heuristic methods perform comparably in terms of speed, and both produce the same scheduling solution for the first experiment. Therefore, solving this scheduling problem validates that both the methods produce optimal solutions.

In the second experiment, the SA obtained the final solution in one millisecond that has optimality. Fig. 20 shows the initial solution from the SA which was infeasible because the schedule violated the deadline constraint for task 4 (see Table 3 for task specifications). Task 4 has a deadline of $t = 11$, whereas its execution in the initial solution finished at $t = 12$. The SA then iterates further and yields the final solution within a millisecond, as shown in Fig. 21. The final solution of the proposed SA produced the same objective value as the exact method. The SA took 1 ms, whereas the exact method took 860 ms to obtain the same solution.

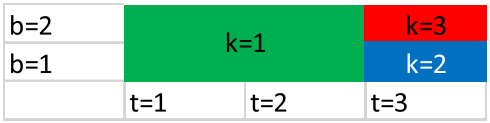

FIGURE 18. Experiment 1 initial solution using SA.

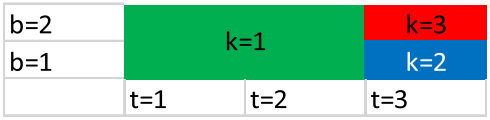

FIGURE 19. Experiment 1 final solution using SA.

Therefore, the second experiment indicates that the heuristic approach outperforms the exact method and finds the optimal solution for scheduling problems of this size.

It was observed that the exact solutions did not scale when the number of tasks increased from 6 to 15 in the third experiment. Therefore, an optimal solution can not be found. Moreover, when the same experiment was conducted using the implemented SA, an initial solution was obtained, as shown in Fig. 22. The SA takes 19 milliseconds to reach the final solution. The objective value, which is 19-time units, is comparable to the exact method solution as shown in see Fig. 23. We further increased the number of tasks to 100 and measured the time required by the SA to obtain the final solution. For this experiment, the exact solution was unable to acquire any feasible solution since the search space with 1000 tasks is exponentially larger.

C. COMPARISON OF QUALITY VS. PERFORMANCE

This section conducts a set of experiments that reflect on the proposed heuristic's quality and performance. Table 5 shows the number of tasks and resources in each experiment for comparing the proposed heuristic's performance and quality against the exact solution. The quality of the solution is related to the makespan objective and is measured in seconds. On the other hand, the performance is the elapsed simulation time to obtain the final solution, measured in milliseconds (ms).

From Fig. 24, we observe that the SA heuristic achieves identical results as the exact method for the first two experiments. This validates that the algorithm can obtain optimal solutions for small-scale problems. The next two experiments provide a suboptimal solution, whereas the exact method fails to carry out the simulation for experiment 4 due to the large search space.

From Fig. 25, we observe that the exact solution method becomes infeasible in experiment 3 as it takes over five minutes to find a solution. Moreover, the method was operated in a CPLEX simulation for more than 20 hours; however, the simulation did not finish, indicating that optimality in the obtained solution is not guaranteed. On the other hand, SA found a near-optimal solution in 19 ms showing an appreciable gain in performance. Moreover, it obtained a

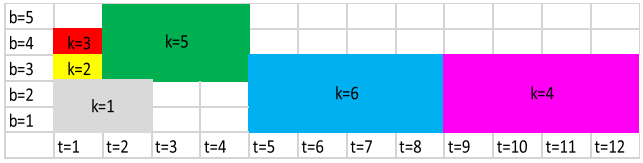

FIGURE 20. Experiment 2 initial solution using SA.

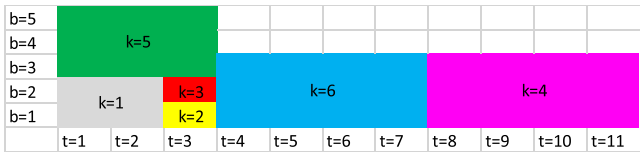

FIGURE 21. Experiment 2 final solution using SA.

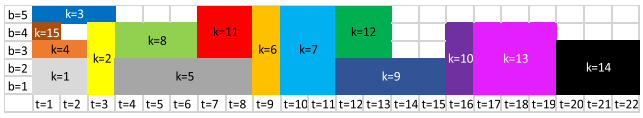

FIGURE 22. Experiment 3 initial solution using SA.

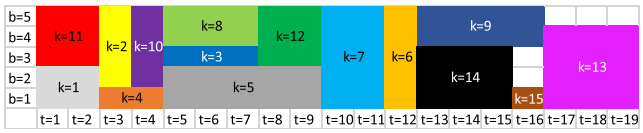

FIGURE 23. Experiment 3 final solution using SA.

near-optimal solution for experiment 4 with 100 resources and 1000 tasks, whereas the exact method failed to run.

In the next set of experiments, we consider the traditional FCFS and SDF algorithms and provide our implementation to draw comparisons between them and the proposed adaptive SA. Table 7 shows the number of tasks and resources used in the three experiments. The incoming tasks were sent to the schedulers in several batches to simulate a real-world cloud environment. Therefore, all the schedulers under the experiment are adaptive and perform dynamic scheduling.

In FCFS, tasks in the list are scheduled in the order they arrive at the data center. The algorithm looks for available resources from the pool of resources and allocates them to each task. In case there are no free resources, it searches for resources that can execute the task at hand before its deadline. As the new set of tasks arrives, the scheduler tries to allocate resources for the first task in the set. It performs a linear search until it finds the required number of available blocks for allocation. It also tracks the estimated time for the busy blocks after which the execution ends. Suppose the scheduler fails to find any available resource in the pool. In that case, it chooses among the occupied blocks that yielded the minimum estimated finish time of execution and queues up the task to these blocks. The estimated finish time of the execution for any block includes the execution time of the currently executed task and tasks previously queued to be executed by the block. Moreover, in contrast to the adaptive SA, if a new set of tasks arrives at the FCFS scheduler, tasks

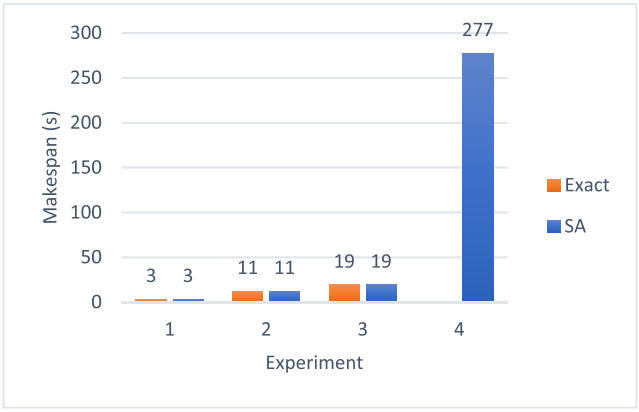

FIGURE 24. Quality (makespan) of exact solution vs. SA.

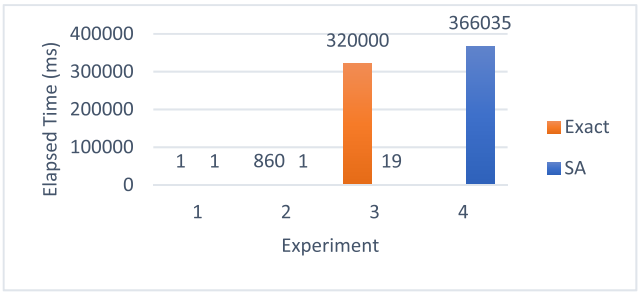

FIGURE 25. Performance (elapsed time) of exact solution vs. SA.

from the old set will not be rescheduled even if the tasks did not start executing. This is to obey the principle of the algorithm so that tasks that come first are always served before others.

In SDF, as a batch of tasks arrives at the data center, the first step in the algorithm is to sort the tasks based on the deadline in ascending order, i.e., shortest first. Then, it follows a similar approach to FCFS when scheduling each task to available resources. However, when SDF deals with a newly arrived list of tasks, it reschedules the tasks from the previous set, unlike FCFS. It considers both the new and previous tasks that did not start execution, and sorts based on deadline before finding a schedule.

To benchmark and compare the quality of solutions achieved by FCFS, SDF and adaptive SA, we use the datasets specified in Table 7. All the three algorithms are implemented and executed against every problem size. Fig. 26 shows for each experiment (x-axis) the makespan that is achieved (y-axis) when all the tasks are scheduled. We can observe that adaptive SA gives the minimum makespan followed by FCFS and lastly SDF. From the graph, it can be concluded that the adaptive SA minimizes the makespan 17% to 28% further compared to FCFS and 17% to 30% further compared to SDF.

Fig. 27 illustrates the time elapsed during each algorithm's simulation to find a schedule. The x-axis represents the problem size, while the y-axis represents the simulation time. FCFS and SDF perform equally or slightly faster than the adaptive SA for the first experiment. For the next two

TABLE 7. Experiments for benchmarking algorithm performance.

Experiment	Tasks	Resources
1	200	50
2	500	100
3	1000	100

experiments, the SA outperforms the other two. The underperformance of FCFS and SDF is because whenever there is a lack of available resources, these algorithms must perform a linear search over all resources until resources that can finish the execution of the selected task before its deadline are found. Performing the search for each task increases in time complexity. In the case of the SA, all the tasks in the list are scheduled at once, validating the deadline constraint without doing any linear search on the resource pool. Therefore, the proposed heuristic technique takes a shorter time to find a schedule. Moreover, SDF takes even longer compared to FCFS because it performs sorting each time before scheduling a new batch of tasks.

D. SENSITIVITY ANALYSIS

In this section, we conduct various experiments to examine the impact of different parameters in the scheduling algorithm.

1) IMPACT OF A VARYING NUMBER OF RESOURCES AND TASKS

The heuristic increases in time complexity when the number of tasks is increased. The degradation in performance is not as much when we increase other parameters, for example, the number of resources in the pool, the average amount of resources required by each task, or the execution deadlines. The impact on the scheduling performance is much greater when the dataset size changes. To demonstrate this, we carried out two series of experiments. In the first series, we decrease the number of resource blocks in each experiment while keeping the number of tasks constant. In the second series, we increase the number of tasks in every experiment while the number of resources remains constant. Both experiments show an increase in the elapsed simulation time. Table 8 and Table 9 show the task specifications of the two experiments.

Fig. 28 plots the results achieved from the experiments specified in Table 8. The x-axis presents the experiment number, and the y-axis presents the elapsed time in milliseconds.

We can observe from the figure that the increase in the elapsed time is linear. Moreover, the increase in elapsed time is because resources must carry out task execution for longer periods of time as we decrease the number of resources in the resource pool. Each resource block must execute a greater number of tasks in series one after another, and a smaller number of tasks are executed in parallel due to a lack of sufficient resources. Therefore, decreasing resources increases simulation time.

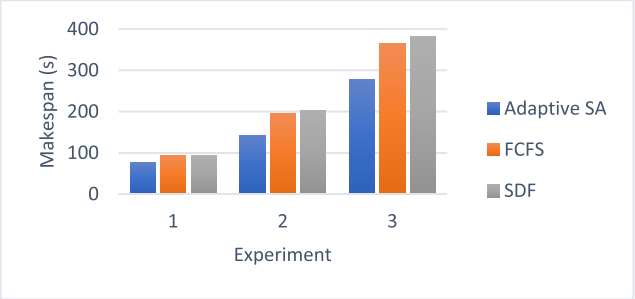

FIGURE 26. Quality (makespan) of adaptive SA vs. FCFS vs. SDF.

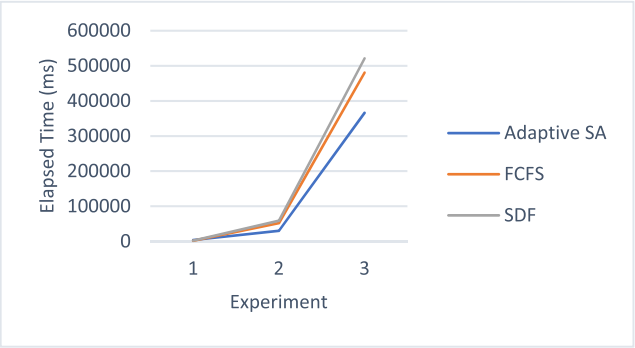

FIGURE 27. Performance (elapsed time) of adaptive SA vs. FCFS vs. SDF.

TABLE 8. Experiments to examine the impact of varying resources.

Experiment	Tasks	Resources	Elapsed Time (ms)
1	500	500	30809
2	500	400	34223
3	500	300	37541
4	500	200	40501
5	500	100	44567

Fig. 29 shows the elapsed simulation time for the experiments in Table 9. The figure shows an exponential increase as we the number of tasks increases, keeping a fixed-size resource pool. The impact of adding tasks to the scheduler is greater than adding more resources. This is because the performance of adaptive SA depends on the complexity equation where K is dominant, as stated in Equation 11.

2) IMPACT OF VARYING MEAN REQUIRED BLOCKS AND EXECUTION TIME

Various cloud tasks require a specific number of resources and execution time. We examine the effect of increasing these two task parameters in the following experiments. On the one hand, the mean required resource blocks are defined as the number of FPGA resource blocks each task needs on average to execute. On the other hand, the mean execution time is defined as the time units a task needs on average to finish execution. Since both are mean values, a Poisson distribution is used to acquire the absolute values for each task.

TABLE 9. Experiments to examine the impact of varying tasks.

Experiment	Tasks	Resources	Elapsed Time (ms)
1	100	500	257
2	200	500	1788
3	300	500	7484
4	400	500	16971
5	500	500	29299

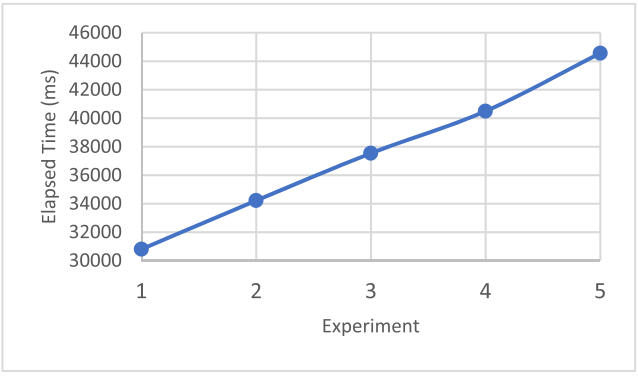

FIGURE 28. Impact of varying resources on elapsed time.

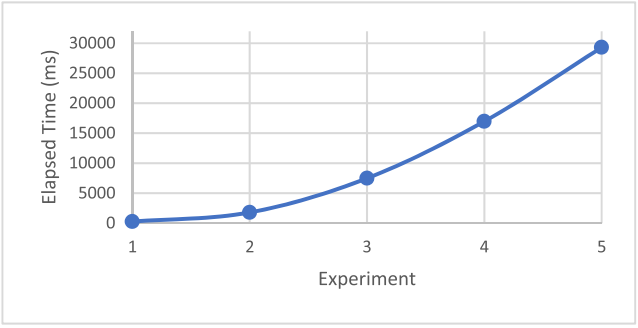

FIGURE 29. Impact of varying tasks on elapsed time.

TABLE 10. Experiments to examine impact of varying mean required resource blocks.

Experiment	Mean Required Resource Blocks	Mean Execution Time (time unit)	Elapsed Time (ms)
1	1	5	12241
2	2	5	14460
3	3	5	20735
4	4	5	29435
5	5	5	44477

Table 10 and Table 11 show the details of the experiments conducted. We set the dataset size to be 500 tasks and the resource pool size to be 500 blocks for all the experiments.

We observe that the increase in both the task parameters shows a slight exponential increase in the elapsed simulation

TABLE 11. Experiments to examine impact of varying mean execution time.

Experiment	Mean Required Resource Blocks	Mean Execution Time (time unit)	Elapsed Time (ms)
1	5	1	9434
2	5	2	15560
3	5	3	21814
4	5	4	29681
5	5	5	44477

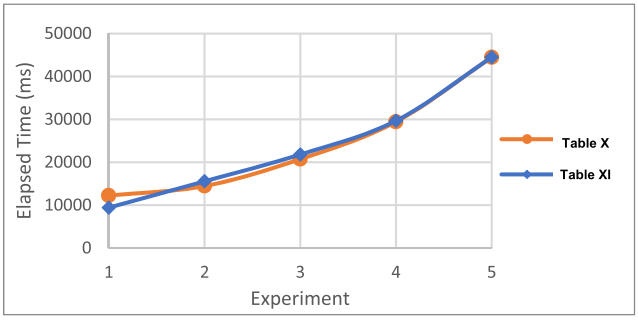

FIGURE 30. Impact of varying mean required resource blocks and execution time on elapsed time.

time in Fig. 30. This observation is because an increase in the mean number of required resources makes tasks in the dataset demand more resources in the FPGA. Since the resource pool has limited resources, more tasks must wait until currently executed tasks are completed. This increases the makespan, as well as the time it takes for the algorithm to schedule all the tasks. Similarly, increasing the mean execution time implies that tasks in the dataset need resources to be allocated for a longer period. Therefore, this parameter has the same impact as the mean required blocks on the elapsed time. In [10] the authors created a test environment in which they processed images using vFPGAs and VMs and compared their results. In this work, we recreated the environment on CloudSim and conducted several simulations. As a result, the execution time for processing an image using a vFPGA is much lesser than using a VM. For example, one of the images finished processing in 0.02310 seconds using a VM. The same image was processed with hardware acceleration in 0.00504 seconds. We can observe that for this test case, by using a vFPGA, the execution time was reduced by about 78%.

VII. CONCLUSION AND FUTURE WORK

Cloud datacenters are rapidly adopting reconfigurable hardware platforms to accelerate the execution of cloud tasks. Moreover, reprogrammability in FPGAs is significant for dynamic requirements in a cloud environment. However, this is not straightforward because the integration of FPGAs in clouds requires sophisticated approaches to provision them as cloud resources. In addition, virtualization is a key feature of the cloud computing paradigm and is essential for FPGA integration. Since FPGA architecture is different from traditional cloud resources, different virtualization mechanisms are developed and proposed in the literature. Besides virtualization schemes, scheduling techniques for FPGA resource pools are also an active field of research. Furthermore, validating virtualization and scheduling approaches using hardware platforms in real-time is not simple. This is because the hardware resources required to set up a cloud environment are expensive, and building the environment is time-consuming. Hence, researchers are heavily dependent on using cloud simulators for validation purposes.

This work explored several FPGA virtualization frameworks and proposed an efficient virtualization approach to DPR-enabled FPGAs in the cloud. Our framework abstracted physical FPGA chips into a pool of PRRs using grid-style partitioning and implemented the MFMA virtualization. Consequently, this enabled multi-tenancy and improved FPGA resource utilization. Moreover, this work used an infrastructure where FPGAs in the hardware layer are connected to host machines via PCIe and network devices via Ethernet. This is unlike most of the works reported in the literature, where only one type of physical connection was established with the FPGA. As a result of additional connectivity, the FPGAs could be used as both local and global accelerators across the network. In addition, the framework used an adapter interface which served as a static communication interface between accelerators and the various framework managers. By automatically generating the adapter, users were allowed to be more productive and focus on application development instead of designing communication interfaces. Furthermore, the role of an FPGA hypervisor was significant as it provided frontend functions to initialize, operate, and terminate vFPGAs. The unified manager interfaces with these functions to efficiently manage and maintain a pool of FPGA resources. Moreover, the hypervisor implementation in this work is novel because the frontend and backend functions were implemented in separate modules as a vFPGA manager and a configuration manager, respectively. This allowed for a modular framework architecture and made implementing the framework in CloudSim easier.

A typical cloud receives a bulk of user requests to use accelerator services, which are the cloud tasks that must be scheduled efficiently. We formulated an optimization model whose objective was to minimize the makespan of cloud tasks. Unlike the models presented in the literature, the proposed model can be used to optimize resource allocation from a pool of architecturally homogenous resources, irrespective of the resource type. This means the model is suitable for resource types such as PRRs in a grid, columnar slots, and even whole FPGA fabrics in a multi-FPGA infrastructure. Moreover, the proposed implementation of the SA algorithm, a metaheuristic technique, yielded suboptimal solutions. We improved the SA by incorporating steps of the Metropolis algorithm, which explored neighboring solutions iteratively. In addition, we developed the SA, making it adaptive to support dynamic scheduling. This was crucial because tasks in a cloud arrive in

real-time instead of in batches. Therefore, the proposed algorithm must be able to dynamically schedule these incoming tasks to FPGA resources. Furthermore, upon performing sensitivity analysis, it was observed that the impact of changing the number of tasks on the simulation time is much higher than changing the number of resources. This was due to the heuristic complexity, which depends on the dataset size as defined by Equation 11. Thus, increasing the total number of tasks leads to exponential growth in the simulation time.

In comparison to the exact method, the SA was validated by experiments using different datasets, where it achieved the same quality of solutions with better performance. It also proved to be scalable even with large datasets, unlike the exact method, which became infeasible when the number of tasks increased from 6 to 15. Moreover, we compared the SA to the FCFS and SDF algorithms using three experiments with different datasets and resource pool sizes. Results showed that the SA minimized the makespan 17% to 28% more than FCFS and 17% to 30% more than SDF.

In future work, we will develop a hardware implementation of the proposed virtualization framework to compare with the software implementation in CloudSim. Moreover, we will consider network constraints and delays in communication between the different CloudSim modules. In addition, the simulator will be included with various FPGA resource schedulers using algorithms such as swarm intelligence, shortest job first, and round-robin. Lastly, we will improve the optimization model to support heterogeneous resource scheduling.

ACKNOWLEDGMENT

This paper represents the opinions of the author(s) and does not mean to represent the position or opinions of the American University of Sharjah.

REFERENCES

[1] (2011). *The NIST Definition of Cloud Computing, SP 800-145*. Accessed: Feb. 28, 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[2] C. C. Editor. *Virtualization Glossary*. CSRC. Accessed: Oct. 30, 2020. [Online]. Available: https://csrc.nist.gov/glossary/term/Virtualization

[3] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualization," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 131–1317.

[4] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proc. 11th ACM Conf. Comput. Frontiers*, May 2014, pp. 1–10.

[5] A. Putnam, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015.

[6] A. M. Caulfield, "Configurable clouds," *IEEE Micro*, vol. 37, no. 3, pp. 52–61, Jan. 2017.

[7] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The Feniks FPGA operating system for cloud computing," in *Proc. 8th Asia–Pacific Workshop Syst.*, Sep. 2017, pp. 1–7.

[8] S. Yazdanshenas and V. Betz, "Quantifying and mitigating the costs of FPGA virtualization," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–7.

[9] A. Al-Aghbari and M. E. S. Elrabaa, "A platform for FPGA virtualization in clouds and data centers," *Microprocess. Microsyst.*, vol. 62, pp. 61–71, Oct. 2018.

[10] A. A. Al-Aghbari and M. E. S. Elrabaa, "Cloud-based FPGA custom computing machines for streaming applications," *IEEE Access*, vol. 7, pp. 38009–38019, 2019.

[11] D. Koch, "Partial reconfiguration on FPGAs in practice—Tools and applications," in *Proc. ARCS*, Feb. 2012, pp. 1–12.

[12] A. Wadhonkar and D. Theng, "A survey on different scheduling algorithms in cloud computing," in *Proc. 2nd Int. Conf. Adv. Electr., Electron., Inf., Commun. Bio-Inform. (AEEICB)*, Feb. 2016, pp. 665–669.

[13] H. Ibrahim, R. O. Aburukba, and K. El-Fakih, "An integer linear programming model and adaptive genetic algorithm approach to minimize energy consumption of cloud computing data centers," *Comput. Electr. Eng.*, vol. 67, pp. 551–565, Apr. 2018.

[14] A. Ealiyas and S. P. J. Lovesum, "Resource allocation and scheduling methods in cloud—A survey," in *Proc. 2nd Int. Conf. Comput. Methodolog. Commun. (ICCMC)*, Feb. 2018, pp. 601–604.

[15] H. Hassan, M. A. Ashraf, W. Hussain, M. S. Akram, A. H. Butt, and Y. D. Khan, "A survey about efficient job scheduling strategies in cloud and large scale environments," in *Proc. Int. Conf. Innov. Comput. (ICIC)*, Nov. 2019, pp. 1–6.

[16] T. A. Xavier and R. Rejimoan, "Survey on various resource allocation strategies in cloud," in *Proc. Int. Conf. Circuit, Power Comput. Technol. (ICCPCT)*, Mar. 2016, pp. 1–4.

[17] M. Alaei, R. Khorsand, and M. Ramezanpour, "An adaptive fault detector strategy for scientific workflow scheduling based on improved differential evolution algorithm in cloud," *Appl. Soft Comput.*, vol. 99, Feb. 2021, Art. no. 106895.

[18] S. A. Murad, A. J. M. Muzahid, Z. R. M. Azmi, M. I. Hoque, and M. Kowsher, "A review on job scheduling technique in cloud computing and priority rule based intelligent framework," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 6, pp. 2309–2331, Jun. 2022.

[19] G. Dai, Y. Shan, F. Chen, Y. Wang, K. Wang, and H. Yang, "Online scheduling for FPGA computation in the cloud," in *Proc. Int. Conf. Field-Programmable Technol. (FPT)*, Dec. 2014, pp. 330–333.

[20] Y. Zhao, C. Tian, Z. Zhu, J. Cheng, C. Qiao, and A. X. Liu, "Minimize the make-span of batched requests for FPGA pooling in cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2514–2527, Nov. 2018.

[21] J. P. Orellana, B. Caminero, C. Carrión, L. Tomas, S. K. Tesfatsion, and J. Tordsson, "FPGA-aware scheduling strategies at hypervisor level in cloud environments," *Sci. Program.*, vol. 2016, pp. 1–12, Jun. 2016.

[22] D. Su, C. Wang, L. Du, R. Li, W. Liu, and D. Zhang, "A cooperative method of task scheduling based on FPGA cloud platform," in *Proc. IEEE 7th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Oct. 2019, pp. 447–450.

[23] M. Bertolino, R. Pacalet, L. Apvrille, and A. Enrici, "Efficient scheduling of FPGAs for cloud data center infrastructures," in *Proc. 23rd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2020, pp. 57–64.

[24] P. S. Choudhary and M. S. Ali, "FPGA-based adaptive task scheduler for real time embedded systems," in *Proc. Int. Conf. Res. Intell. Comput. Eng. (RICE)*, Aug. 2018, pp. 1–4.

[25] Z. G. Mohammed, A. M. A. Hamdoon, and M. S. Aziz, "Scheduling lecturer system based on FPGA," in *Proc. Int. Conf. Advance Sustain. Eng. its Appl. (ICASEA)*, Mar. 2018, pp. 54–58.

[26] T. Yu, B. Feng, M. Stillwell, L. Guo, Y. Ma, and J. Thomson, "Lattice-based scheduling for multi-FPGA systems," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 318–321.

[27] S. Deniziak and S. Bak, "Scheduling of distributed applications in HHP-CaaS clouds for Internet of Things," in *Proc. 23rd Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2020, pp. 1–4.

[28] A. Al-Zoubi, K. Tatas, and C. Kyriacou, "Towards dynamic multi-task schedulling of OpenCL programs on emerging CPU-GPU-FPGA heterogeneous platforms: A fuzzy logic approach," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2018, pp. 247–250.

[29] R. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw., Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011.

[30] B. Louis, "CloudSimDisk: Energy-aware storage simulation in CloudSim," M.S. thesis, Dept. Comput. Sci., Elect. Space Eng., Luleå Univ. Technol., Luleå, Sweden, Dec. 2015.

[31] O. Knodel, "FPGAs and the Cloud–An endless tale of virtualization, elasticity and efficiency," *Int. J. Adv. Syst. Meas.*, vol. 11, nos. 3–4, pp. 230–249, Dec. 2018.

[32] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. Comput.*, vol. C-22, no. 4, pp. 421–422, Apr. 1973.
[33] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–7.
[34] *Amazon EC2 F1 Instances*. Amazon. Accessed: Feb. 14, 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
[35] *AWS F1*. Amazon Web Services. Accessed: Feb. 14, 2019. [Online]. Available: https://www.xilinx.com/support/university/aws-f1.html
[36] *FPGAs, Microsoft Project Brainwave and DNN Acceleration*. Intel. Accessed: Feb. 14, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/analytics/artificial-intelligence/accelerating-ai-with-microsoft-project-brainwave.html
[37] *Instance Type Families*. Alibaba Cloud Documentation Center. Accessed: Feb. 14, 2019. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/25378.html?spm=a2c65.11461447.0.0.3f7a2ff9TXWNyX
[38] K. Beghdad Bey, F. Benhammadi, F. Sebbak, and M. Mataoui, "New tasks scheduling strategy for resources allocation in cloud computing environment," in *Proc. 6th Int. Conf. Modeling, Simulation, Appl. Optim. (ICMSAO)*, May 2015, pp. 1–5.
[39] J. Letkowski, "Applications of the Poisson probability distribution," in *Proc. Academic Bus. Res. Inst. Conf.* San Antonio, TX, USA, Mar. 2012, pp. 1–11.
[40] O. Knodel, P. Genssler, and R. Spallek, "Virtualizing reconfigurable hardware to provide scalability in cloud architectures," in *Proc. CENICS 10th Int. Conf. Adv. Circuits, Electron. Microelectron.* Rome, Italy, Sep. 2017, pp. 33–38.

ABID FARHAN received the B.Sc. and M.Sc. degrees in computer engineering from the American University of Sharjah, in 2017 and 2019, respectively. He worked as a Graduate Teaching Assistant and a Research Assistant with the American University of Sharjah. His research interests include field-programmable gate array and cloud computing. He is a member of the Upsilon Pi Epsilon Honor Society. He won several awards, including the Chancellor's Scholarship and the Sharjah Islamic Bank Annual Research Award.

RAAFAT ABURUKBA received the bachelor's degree in computer science and software engineering, and the master's and Ph.D. degrees in computer engineering from the University of Western Ontario, London, ON, Canada. He is currently an Assistant Professor with the Computer Science and Engineering Department, American University of Sharjah (AUS). Before joining AUS, he was a Faculty Member of the Computer Science and Software Engineering Department, Pennsylvania State University, Erie, Pennsylvania. His research interests include cloud computing middleware and applications, fog computing, cooperation and coordination in distributed systems, privacy in distributed systems, economic-based models and approaches for decentralized scheduling, and smart spaces.

ASSIM SAGAHYROON (Senior Member, IEEE) received the M.Sc. degree in electrical engineering from Northwestern University, Evanston, IL, USA, and the Ph.D. degree from The University of Arizona, Tucson, AZ, USA. From 1993 to 1999, he was a member of the Department of Computer Science and Engineering, Northern Arizona University. In 1999, he joined the Department of Mathematics and Computer Science, California State University. In 2003, he joined the Department of Computer Science and Engineering, American University of Sharjah, where he served as the Department Head for seven years. He was an Invited Technical Reviewer for some of the National Science Foundation programs and served on the technical program committees of many conferences. He has many publications in international conferences and journals. His research interests include innovative applications of emerging technology in the medical field, power consumption of portable electronics, and FPGAs-based design.

MOHAMMED ELNAWAWY received the B.Sc. *(summa cum laude)* and M.Sc. degrees in computer engineering from the American University of Sharjah, United Arab Emirates, in 2017 and 2019, respectively. In 2020, he joined as a Laboratory Instructor with the Computer Science and Engineering Department, the American University of Sharjah, where he worked as a Graduate Teaching and Research Assistant. He won several awards, including the American University of Sharjah Graduate Student Research, Scholarly, and Creative Work Excellence Award, and the Sharjah Islamic Bank Research Award. He is a member of the Upsilon Pi Epsilon Honor Society. His research interests include machine learning, field-programmable gate arrays, and embedded systems.

KHALED EL-FAKIH is currently a Professor with the College of Engineering, American University of Sharjah, where he joined in September 2001. He spent a sabbatical year at the Verimag Laboratory, University of Grenoble 1 (UJF), France. He acted as the Program Co-Chair of the 2008 International Conference on Formal Techniques for Networked and Distributed Systems and 2015 International Conference on Testing Software and Systems. His research interests include formal testing, automatic synthesis of distributed systems, optimization, and application of heuristic algorithms.