# Spatiotemporal Strategies for Long-Term FPGA Resource Management

Atefeh Mehrabi
*Duke University*
atefeh.mehrabi@duke.edu

Daniel J. Sorin
*Duke University*
sorin@ee.duke.edu

Benjamin C. Lee
*University of Pennsylvania*
leebcc@seas.upenn.edu

***Abstract*—The deployment of increasingly large and capable FPGAs has motivated mechanisms for sharing them, but system support for FPGAs is not yet mature. Traditional scheduling algorithms do not account for the unique characteristics of FPGAs, leading to infeasible or inefficient allocations. We propose a novel scheduling policy, called Spatiotemporal FPGA Scheduling, that overcomes these challenges to achieve long-term target allocations by tracking and correcting deviations from targets across management time periods. Compared to traditional algorithms, Spatiotemporal FPGA Scheduling produces allocations that are up to 32% closer to targets, improves average throughput by up to 44%, and improves average FPGA utilization by up to 23%.**

***Index Terms*—FPGA, resource management, scheduling, spatial sharing, temporal sharing, partial reconfiguration**

## I. Introduction

The deployment of increasingly large, capable FPGAs has motivated mechanisms for sharing them. First, vendors enable FPGA virtualization to increase efficiency and amortize capital costs. Hardware-based virtualization supports running multiple, concurrent applications by partitioning an FPGA's physical resources into slots [1], [2], [3], [4]. Using one or more slots to define a virtual FPGA is efficient when an application demands only a fraction of the FPGA.

Second, modern FPGAs support partial reconfiguration (PR), which programs a subset of the FPGA's resources. One FPGA slot can be re-programmed while applications on other slots run uninterrupted [1], [2], [5], [4], [6], [7], [8]. Partial reconfiguration improves flexibility as slots can be allocated to applications on demand, and it is faster than programming the entire FPGA [9].

**Challenges in FPGA Management.** Despite recent advances, efficient and responsive FPGA management remains difficult even for a single FPGA. Applications often underutilize slots and other applications cannot derive utility from slots' excess resources. Virtualization enables *spatial sharing* but falls short of its promised efficiency because fine-grained resources are organized into coarse-grained slots. Resources are often stranded in slots because FPGA demands and allocations are defined in terms of minimum, indivisible bundles.

Partial reconfiguration facilitates *temporal sharing* but only up to a point. Unlike in general-purpose processors, context switches in FPGAs are slow and unwieldy [10], [11]. The FPGA must checkpoint the running application and incur the costs of extracting internal hardware state distributed across CLBs, BRAMs, and DSPs [9]. The FPGA can instead wait for the running application to arrive at some minimal state with fewer elements to store [12]. Nonetheless, transition costs preclude frequent preemptive scheduling.

These challenges hinder our goal: efficiently allocating a shared FPGA's resources to meet the system's *desired target allocations*. Target allocations could be dictated by priorities associated with workload importance or entitlements associated with external factors (*e.g.*, users' financial contributions to a shared system). In this paper, without loss of generality, our target allocations are based on max-min fairness.

We propose Spatiotemporal FPGA Scheduling (STFS) to consistently achieve target allocations in virtualized, multi-tenant FPGAs. Aware of FPGA characteristics, the policy favors providing logical shares when possible and exploits temporal scheduling when necessary due to gaps between logical shares and physically feasible ones. Within short time periods, STFS allows deviations from logical shares in the spatial domain when these deviations favor throughput, improve utilization, or avoid preemption. STFS adjusts future shares to compensate for past deviations from logical shares such that each application achieves its target allocation, on average, across time. In sum, we make the following contributions.

- We survey FPGA technology to identify current limitations and emerging capabilities. We identify unique FPGA characteristics that impact their management.
- We propose STFS, which tolerates short-term deviations from target allocations to improve throughput and utilization. It achieves long-term target allocations by tracking historical decisions and compensating for past deviations.
- We demonstrate STFS's ability to respond to system dynamics such as application arrivals and departures.

We consider multiple *applications* that seek to instantiate *accelerators* on a shared FPGA. For each application, a stream of *tasks* arrive over time. Think of tasks as requests for each pre-implemented accelerator service. We are motivated by clouds that deploy FPGAs [13], [14] and offer pre-implemented accelerators as services to users' virtual machines [13], [15], [16].

System throughput increases with the number of tasks served from each application. The FPGA hosts multiple accelerators simultaneously and must manage shared resources. *Allocation* and *scheduling* refer to management decisions made in space and time, respectively. Informally, we use allocation and scheduling interchangeably because STFS takes

an integrated spatiotemporal approach to pursue throughput for accelerated tasks on shared FPGAs.

## II. Emerging Field Programmable Gate Arrays

FPGAs are evolving in response to their emerging role in high-performance systems. They can now communicate with host processors via cache-coherent shared memory [17], [18], which mitigates a communication bottleneck [19] and encourages the design of fine-grained accelerators [20] This section surveys recent advances in FPGAs and the next discusses how technology constraints impact run-time management.

### A. Reconfigurable Logic and Slots

An FPGA is a pool of connected, reconfigurable logic elements: look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), digital signal processors (DSPs), and routing resources. An FPGA organizes its logic elements in columns such that each column contains one type of logic (*e.g.* DSP) [7], [21]. In commercial FPGAs, such as the Xilinx Zynq, columns may not repeat with a consistent pattern. Vertically aligned reconfigurable cells are typically homogeneous, except where I/O blocks are located, but horizontally aligned ones are not. FPGA regions with the same number of rows and columns may differ in the number of LUTs, FFs, *etc.*

**Slots.** An FPGA's pool of logic may be partitioned into a number of reconfigurable regions, called slots, to support multi-tenancy. A slot is a rectangular region containing some fraction of the FPGA's logic [7]. The specific number of reconfigurable resources in a slot depends on its size and location within the FPGA. Each slot can be reconfigured independently to implement multiple accelerators on an FPGA [22].

Static slot sizes permit ahead-of-time compilation [5], especially when FPGA needs are well established (*e.g.*, pre-implemented accelerator libraries) or can be anticipated as tasks flow through queues (*e.g.*, OpenCL work groups) [9]. However, static slot sizing presents challenges in a setting with diverse accelerators. Larger slots risk poor utilization and unexploited concurrency when accelerators are small. Smaller slots may require combining multiple slots, when accelerators are large, which is not easily supported by EDA tools [23] and complicates FPGA mappings.

**Shell.** A portion of the FPGA is reserved for the shell, which provides infrastructure for communicating with system components [24], [23], [25]. The shell consumes 20-50% of its logic and reduces its capacity for accelerators [23], [26].

### B. Compilation and Reconfiguration

Accelerator design might start by compiling a design written in a high-level language into a hardware description at the register-transfer level [27], [28]. Next, the hardware description is synthesized into a netlist. Finally, the netlist is mapped to the FPGA's physical hardware by placing logic and routing interconnect. The resulting bitstream encodes information necessary to program and prepare the FPGA for execution.

Instantiating an accelerator on an FPGA requires time for synthesis, place-and-route, and programming. Synthesis requires minutes to generate a netlist whereas place-and-route could require hours to map the netlist to the fabric [5], [29], [30]. These costs are typically incurred offline, amortized over long periods of computation, and rarely impact online performance [5].

Programming the FPGA with a bitstream requires tens to hundreds of milliseconds [21], [7], a relatively small cost but one that directly impacts allocation responsiveness. Programming time decreases with the bandwidth of the configuration port (*i.e.*, ICAP). Emerging non-volatile or stacked memories might expand the FPGA's bitstream caches and reduce its demand for bandwidth. Programming time increases with bitstream size. Partial reconfiguration can reduce bitstream size by targeting only a subset of the FPGA's slots [31], [9].

Partial reconfiguration modifies the accelerator in an FPGA slot without interrupting other running accelerators and without affecting the shell. Designers can reconfigure at scales as large as the entire FPGA or as small as an individual resource (*e.g.*, LUT). Finer granularities increase flexibility and utilization but impose architectural costs. Xilinx FPGAs can reconfigure slots and address regions as small as one element wide and one clock region tall [21], [7].

## III. Management Challenges for Shared FPGAs

Shared systems amortize hardware costs over more users and power costs over more computation. Yet strategic users require incentives, often rooted in promised shares calculated from some sort of agreement (*e.g.*, priorities, entitlements,...) if they are to forgo private systems in favor of shared systems.

Shared FPGAs present yet another setting where meeting target allocations is critical *but conventional policies are insufficient*. Virtualization and partial reconfiguration provide mechanisms for sharing FPGAs but not managing them. One might think policies for other computing resources (*e.g.*, processor cores, cache and memory) could be simply deployed for FPGA resources. But our review of existing allocation policies reveals limits given the unique characteristics of FPGAs.

Specifically, one-shot policies have been used to divide resources in chip multiprocessors and data centers [32], [33], [34], [35], [36]. Such policies pursue target allocations with repeated, independent allocation decisions in each time period. Allocations are independent such that decisions in the present have no effect on those in the future. Examples of such policies focused on fairness include max-min, dominant resource [35], and resource elasticity fairness [34].

However, applying one-shot policies for FPGA management is challenging. Policies may calculate and prescribe logical allocations that require infeasible or inefficient physical allocations due to unique FPGA characteristics. In this section, we describe constraints that arise from FPGAs' utility functions, restrictive slots, and context switches.

### A. Step-function Utility

A bitstream is specifically compiled for RTL on a particular slot. An application derives utility and runs its bitstream successfully only on the slot it has been compiled for. The

host slot must provide enough reconfigurable resources to meet the compiled application's demand. Otherwise the application derives no utility.

Suppose 3 applications each demand 3 slots, but the FPGA has only 6 slots. Imagine a one-shot policy that targets max-min fair allocations. Max-min divides resources fairly among the 3 applications with identical weights and assigns 2 slots to each application. When these logical shares are mapped to physical slots, no application can instantiate its accelerator and derive utility.[1]

The FPGA's step-function utility contrasts with those for processor cores, cache, memory, and communication bandwidth. For these conventional resources, which are invariably time multiplexed, applications derive partial utility even when allocations are smaller than demands without further effort [34]. Prevalent policies, such as max-min or resource elasticity fairness for conventional resources, assume partial utilities when calculating applications' shares of scarce resources. FPGAs, which do not easily produce partial utilities, must look beyond traditional policies.

### B. Restrictive FPGA Slots

The practicality of logical allocations depends on how physical resources are organized into slots, which raises several difficulties. First, policies' logical allocations are fractional and often calculated to arbitrary levels of precision [35], [34], but FPGAs' physical resources are organized into fixed size slots. This mismatch leads either to wasted resources within a single slot or awkward partitions across multiple slots.

Second, designing the FPGA with a few large slots leads to poor utilization when accelerators are small. Rounding fractional allocations or using one-shot allocation policies for indivisible goods [36] under-utilizes slots and strands resources. Recovering these resources requires dividing a slot for multiple accelerators, but such division requires merging designs into a single bitstream, preventing independent partial reconfiguration of colocated accelerators.

Third, designing the FPGA with many small slots might leave some slots unallocated due to mapping constraints. Small slots might be combined for large accelerators but, to avoid routing complexity, current tools combine only adjacent slots [23], [37].

Finally, an FPGA's slots are not fungible due to the heterogeneous layout of reconfigurable resources across the fabric's columns. Resources (*e.g.*, LUTs) can be distributed differently in different slots, which means an accelerator's bitstream targets a specific slot and multiple bitstreams are required to target different slots [23]. Accelerators cannot easily migrate from one slot to another at run-time with standard toolflows [38], [37].

[1]One could imagine deriving partial utilities from allocations smaller than demands, either by (a) partitioning the design into phases and time multiplexing slots, or (b) making smaller accelerators that fit the slot size (e.g., by parameter tuning [28]). These options, however, have serious drawbacks.

**Algorithm 1** Simplified STFS Algorithm

```
1: S → total number of FPGA slots
2: S_idle → number of idle slots
3: n → number of apps
4: D[i] → number of slots demanded by app i, i∈[1,n]
5: success[i] → success rate for app i
6: CurrAlloc[i] → number of slots currently allocated to app i
7: TotalAlloc[i] → sum of slots allocated to app i
8: AvgAlloc[i] → mean of slots allocated to app i
9: TargetAlloc[i] → logical share for app i
10: interval = 0
11: TotalAlloc[i] = 0 for all i
12: for each time interval do
13:     interval++
14:     S_idle = S
15:     *update AvgAlloc[i] and success[i]   for all i*
16:     CurrentAlloc[i] = 0 for all i
17:     while S_idle > Min{D[i]|i∈[1,n]} do
18:         next ← i with smallest success[i]
19:         if S_idle > D[next] then
20:             CurrAlloc[next] += D[next]
21:             TotalAlloc[next] += D[next]
22:             AvgAlloc[next] = TotalAlloc[next] / interval
23:             success[next] = AvgAlloc[next] / TargetAlloc[next]
24:             S_idle -= D[next]
25:         else
26:             *don't choose i again in this interval*
```

### C. Expensive Context Switches

Context switches are very expensive because the computation's state—distributed across fine-grained LUTs, RAMs, *etc.*—must be extracted, saved, and restored [39]. The FPGA can extract state using scan chains, which instrument all memory elements via a shift register mode, but this requires milliseconds [10], [40], [11]. It can save state using direct memory accesses, but this requires logic for the DMA engine [41]. To reduce the amount of state that must be extracted, context switches can wait for task-specific consistency points (*e.g.*, OpenCL work groups) [12], [9]. Moreover, only one FPGA slot can be reconfigured at a time. Because context switches are so expensive for FPGAs, prior work on scheduling processor cores and memory bandwidth [42], [43] does not translate easily to reconfigurable fabrics.

## IV. Spatiotemporal FPGA Scheduling

Spatiotemporal FPGA Scheduling (STFS) is an iterative, interval-based scheduling algorithm that uses past success in maintaining target allocations to guide current allocations. The system defines the target allocation (logical share) for each application. Without loss of generality, max-min fairness defines our targets. STFS defines `success rate` to measure

| Time | CurrAlloc A | CurrAlloc B | CurrAlloc C | AvgAlloc A | AvgAlloc B | AvgAlloc C | Success Rate A | Success Rate B | Success Rate C | $S_{idle}$ | Final Allocation |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** | 0 | 0 | 6 | A, A, A, B |
| | 1 | 0 | 0 | 1 | 0 | 0 | 0.5 | **0** | 0 | 5 | |
| | 1 | 3 | 0 | 1 | 3 | 0 | 0.5 | 1.5 | **0** | 2 | |
| | 2 | 3 | 0 | 2 | 3 | 0 | 1 | 1.5 | **0** | 1 | |
| | 3 | 3 | 0 | 3 | 3 | 0 | 1.5 | 1.5 | **0** | 0 | |
| T1 | 0 | 0 | 0 | 1.5 | 1.5 | 0 | 0.75 | 0.75 | **0** | 6 | A, A, C |
| | 0 | 0 | 4 | 1.5 | 1.5 | 2 | **0.75** | 0.75 | 1 | 2 | |
| | 1 | 0 | 4 | 2 | 1.5 | 2 | 1 | **0.75** | 1 | 1 | |
| | 2 | 0 | 4 | 3 | 1.5 | 2 | 1.5 | **0.75** | 1 | 0 | |

TABLE I: STFS schedules A, B, C with demands 1, 3, 4 slots on six FPGA slots. Iterative updates for two intervals.

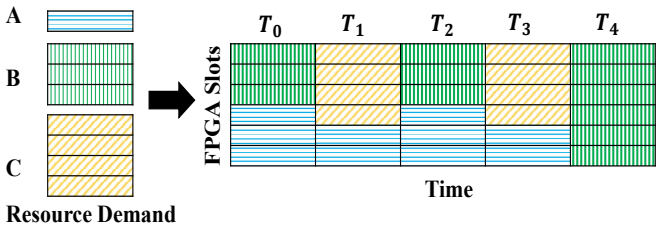


Fig. 1: STFS schedules A, B, C with demands 1, 3, 4 on six FPGA slots. Slot allocations for five intervals.

how closely an application's physical allocations of slots across time match its logical share.

$$\text{Success Rate} = \frac{\text{Average \# Allocated Slots in Prior Intervals}}{\text{Target Slot Allocation}}$$

Applications with lower success rates have been treated more unfairly with respect to their target allocation in the past and are prioritized for future scheduling. The algorithm seeks to ensure applications receive target allocations in the long run.

### A. Scheduling Algorithm

Algorithm 1 presents STFS. In each interval, each slot is allocated to only one application and some applications may be allocated multiple slots. If a task completes well before the end of an interval, the accelerator may complete multiple tasks. If a task is incomplete at the end of an interval, the task is permitted to complete and avoid preemption.

In each interval, STFS iteratively selects the application with the next highest priority (Line 18). The selected application will map one instance of its accelerator if the number of idle FPGA slots suffices (Line 19). Otherwise, it will be skipped. Then, the algorithm updates the count of remaining slots as well as applications' average allocations, success rates, and priorities (Lines 20-24). When all slots are used or remaining slots are insufficient for any application, the algorithm terminates and accelerators are instantiated in slots.

When the application with the highest priority does not fit in the remaining idle slots, STFS selects the application with the next highest priority that fits (if any). This allocation, while unfair with respect to target allocations in the moment, improves throughput and avoids under-utilization. The deviation from target allocations for the skipped application is reflected in its lower success rate and is compensated in future intervals.

The algorithm addresses two challenges in FPGA scheduling. First, it uses non-preemptive scheduling to avoid frequent context switches. Second, STFS reconfigures at regular intervals and reallocates all slots together, allowing the scheduler to optimize the mix of colocated applications. In contrast, releasing and re-allocating slots at irregular intervals (*e.g.* as heterogeneous tasks complete) would increase the likelihood of idle slots as slots cannot be used until there are a sufficient number of slots to meet an application's demand.

### B. Example Operation

Suppose the FPGA is partitioned into 6 identical slots, and 3 applications (A, B, C) demand 1, 3, and 4 slots, respectively.

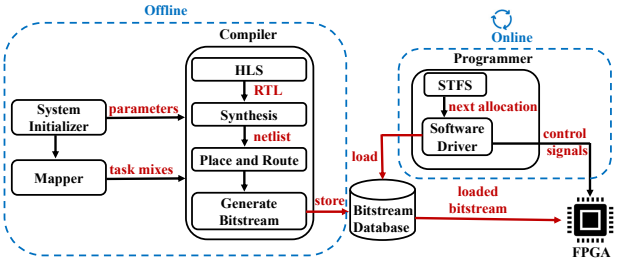


Fig. 2: STFS integrates with FPGA compilation and programming flow.

Without loss of generality, suppose each application's target allocation is 2 slots. For the first two iterations, Table I shows allocations, available resources, and success rates. In each row, the application with the smallest success rate (bold) has the highest priority for allocation.

In interval $T_0$, applications have identical priorities. A and then B are granted resources in order. C has the next highest priority and requires 4 slots, but only 2 idle slots remain. Here, STFS deviates from target allocations and allocates the remaining slots to A rather than leaving them idle. C carries its deficit from $T_0$ into $T_1$ via its low success rate. In $T_1$, C is the first task to receive allocations. After C and then A receive slots, B has the next highest priority but does not fit into remaining slots. As in $T_0$, A receives these slots rather than leaving them idle.

Figure 1 illustrates slot usage over time and highlights a new scenario in $T_4$. B's priority has increased due to multiple deviations from target allocations in past intervals, making it eligible to receive two consecutive allocations in one interval.

### C. FPGA Compilation and Programming

Figure 2 shows how STFS is integrated into the FPGA's compilation flow and our software toolchain.

**Initializer.** Initialization configures parameters in Algorithm 1. In the space domain, parameters include the size, shape, location, and total number of reconfigurable slots. In the time domain, interval length is the key parameter. The interval should be long enough to amortize the latency of partial reconfiguration, which can be modeled as $P_s \times S/P_t$ where $P_s$ is a slot's bitstream size, $S$ is the number of allocated slots, and $P_t$ is the configuration port throughput [9], [21]. Initialization also includes a reference one-shot policy (*e.g.*, max-min fairness)

or numerical values that specify applications' target allocations against which success rates are calculated.

**Mapper.** Given a set of applications (*e.g.,* datacenter acceleration services), the Mapper generates a set of possible application mixes and FPGA mappings, accounting for FPGA capacity and mapping constraints. For example, in Figure 1, 6 slots could be used to host either 6 type-A accelerators or 2 type-A and 1 type-C accelerator, *etc.*

Each colocated mix can be mapped to FPGA slots in multiple ways. When accelerator A and C share 6 slots, in theory, A could occupy any 2 slots on the FPGA and leave 4 slots for C, creating $\binom{6}{2} = 15$ logical mappings. In practice, however, a feasible mapping requires 2 and 4 adjacent slots for A and C, respectively (Figure 1 at $T_1$). The Mapper picks a feasible mapping for compilation. Any other feasible mapping with similar application mix exploits bitstream relocation at runtime and does not require re-compilation [37], [44].

**Compiler.** The compiler produces bitstream(s) for the accelerator(s) and the logic that integrates them with the rest of the FPGA. For every mapping, the compiler produces a full bitstream, which programs the entire FPGA. The compiler also produces partial bitstreams for individual accelerators within each application mix.

Accelerator compilation can start directly from the task's RTL or use high-level synthesis (HLS) tools to automatically generate RTL for a task written in a high-level language (*e.g.*, C/C++). Vivado HLS generates accelerator IPs wrapped with AXI interconnects and assigns address spaces to memory-mapped control planes [23]. Vivado Design Suite integrates the accelerators with the rest of the FPGA.

Multiple application mixes are compiled and their bitstreams are cached for run-time deployment, as in prior works [5]. Compiling application mixes is practical for several reasons. First, for libraries of cloud accelerators, data center operators know applications well in advance, allowing them to compile accelerators offline and load cached bitstreams based on users' online demands. Second, compilation for multiple application mixes is highly parallelizable. Third, research in FPGA place-and-route and programming has reduced compilation latency significantly$\times$ [45], [46], [47] or produced strategies to hide the latency [29], [48].

**Programmer.** When STFS re-allocates slots to applications, the FPGA is reprogrammed with a bitstream from the cache. Xilinx tools use embedded software drivers to boot the board with required settings and establish communication channels between the shell and hardware accelerators [49], [50].

## V. Evaluation

**Platform.** We evaluate STFS on a Xilinx Zedboard XC7Z020 from the Zynq-7000 SoC family. We create one column with $S \in [3, 6, 12]$ vertically aligned slots on a portion of the programmable logic. Because our device requires 1.1 ms to configure all slots, we set the allocation interval to be 1.1 seconds and thus ensure reconfiguration overhead is less than 0.1%.

| Benchmark | $\mathbf{S=3}$ | $\mathbf{S=6}$ | $\mathbf{S=12}$ |
|---|---|---|---|
| AES | 1 | 1 | 2 |
| BFS | 1 | 1 | 1 |
| SHA | 1 | 1 | 2 |
| SPMV | 1 | 2 | 3 |
| GSM | 1 | 2 | 4 |
| FFT | 2 | 3 | 5 |
| SORT | 3 | 5 | 10 |
| VITERBI | 3 | 5 | 10 |

TABLE II: Benchmarks and their demands for slots. FPGA is organized into 3, 6, and 12 slots. Slots under S=3 contains (BRAM, DSP, FF, LUT) = (30, 40, 20800, 10400). Slots under S=6 and S=12 are 2 and 4 times smaller, respectively.

**Workloads.** We evaluate STFS with benchmarks that exhibit diverse resource demands and run-to-completion times. Table II lists eight benchmarks from two suites used in prior FPGA accelerator studies, MachSuite [52] and CHStone [5]. We combine benchmarks to create two workloads. The *micro-workload* includes AES, GSM, FFT, and VITERBI. This small subset of accelerators allows us to visualize, understand, and evaluate detailed allocation dynamics for each accelerator. The *full-workload* includes all accelerators. The workload generates a stream of computational tasks for each accelerated application. Table II details the number of slots demanded by each benchmark under different slot sizes. Note that FPGA and accelerator sizes do not affect our evaluation, which focuses on how diverse accelerators share an FPGA.

**Baselines.** We compare against variants of round-robin: Plain Round-Robin (PRR), Relaxed Round-Robin (RRR), and Deficit Round-Robin (DRR). PRR has been used in prior FPGA work [20] but RRR and DRR have not.

*"Plain Round-Robin (PRR)"* is the traditional ordering policy. In the current interval, allocation begins with the application that did not receive its allocation in the previous interval. Allocation proceeds in order until an application's demand exceeds remaining capacity. This variant strictly follows round-robin order even if resources remain unallocated.

*"Relaxed Round-Robin (RRR)"* is a policy we created to improve Plain Round-Robin. If the next application in round-robin order cannot fit its request within the remaining slots, those slots will be allocated to the next application that can fit. In each interval, RRR first allocates for applications that did not receive resources when it was their round-robin turn due to capacity constraints. RRR then continues in round-robin order. Thus, RRR improves resource utilization at the cost of temporarily violating round-robin order.

*"Deficit Round-Robin (DRR)"* has been used in packet scheduling [53], [54] but not FPGAs. For each application, DRR maintains a deficit counter that tracks accumulated deviations between actual and target allocations. DRR initializes the counter to zero, increments the counter by the application's target allocation at the beginning of each time interval, and decrements the counter based on the number of allocated slots. Resources are allocated in round-robin order and an application receives allocations only if its counter value is greater than its request for slots. If an application requests but

| Prior Work | Multi-Tenancy | Dynamic Reconfiguration | Scheduling Policy | Scheduling Limitation |
|---|---|---|---|---|
| ViTAL [30] | yes | yes | simple matching between requests and capacity | lacks spatiotemporal fairness |
| OPTIMUS [20] | yes | no | round-robin | lacks spatial fairness |
| Vaishnav et al. [9] | yes | yes | Jain Index | lacks temporal fairness |
| AmorphOS [5] | yes | yes | equal partitioning of IO and bandwidth | lacks spatiotemporal fairness |
| Coyote [39] | yes | yes | round-robin | lacks spatial fairness |
| Blaze [51] | yes | no | first-come-first-serve | lacks spatiotemporal fairness |
| Fahmy et al. [3] | yes | yes | no policy, rejects requests when FPGA resources are unavailable | lacks spatiotemporal fairness |

TABLE III: Prior work on virtualizing shared FPGAs neglect schedulers that allocate according to targets in space and time.

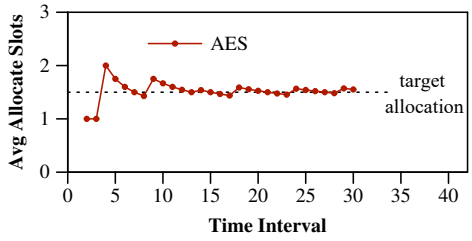


Fig. 3: Average slot allocations on FPGA with six slots. STFS schedules *micro-workload*, which includes AES. Demand for one accelerator instance of AES is 1 slot.

does not receive slots due to capacity and counter constraints, it carries unused counters into future intervals.

Round-robin is a natural baseline for allocating resources across time. Indeed, prior work provides few better alternatives. Many existing systems have deployed single-tenant FPGAs [55], [13]. Table III also indicates that prior studies in virtualizing shared FPGAs have neglected the spatiotemporal scheduling question. Their schedulers do not consistently satisfy allocation targets, neglect wasted slots, and incur preemption costs. Ours is the first to allocate FPGA slots, permitting short-term deviations yet achieving long-term targets.

### A. Aligning with Target Allocations

**Long-term Allocations.** Figure 3 shows the average slot allocations for one benchmark, AES, while STFS is scheduling *micro-workload*. The target is 1.5 slots per application, and the figure shows that allocations may temporarily deviate from the target in the short term (see spikes in early intervals), but STFS tracks these deviations to guide future allocations and ensure average allocations align with the target in the long term.

Figure 4 summarizes slot allocation over time for *micro-workload*. RRR, which has no memory of past allocations, produces large deviations between the realized and target allocations. Large accelerators, like VITERBI, dominate and receive considerably more slots over time. PRR, not shown, has similar allocation dynamics but performs worse. DRR's allocations more closely align with target allocations when compared to PRR and RRR, but still incur unbalanced success rates from 86% to 100%. STFS's allocations achieve 100% success rates for all applications, with better slot utilization and faster convergence.

Figure 5 repeats the analysis for *full-workload* and presents similar outcomes. The target is 0.75 slots for each of eight accelerators. STFS successfully and quickly converges to this target for all accelerators. In contrast, other policies suffer from spatially unfair allocations with unbalanced success rates relative to target allocations. These other policies also require more time to converge to targets or waste resources.

Figure 6 demonstrates average success rates among applications within *full-workload*. STFS outperforms baselines by converging to target allocations. Its success rates are on average 32%, 17%, and 9% higher than those from PRR, RRR, and DRR, respectively. Even DRR, the best of the baselines, cannot align its allocations with targets.

**Short- versus Long-Term Allocations.** There exists a trade-off between strictly enforcing allocation targets in every time interval and flexibly aligning with targets across many time intervals, the strategy pursued by STFS. Figure 7 explores this trade-off by constraining how often STFS is permitted to deviate from a logical share. When no deviations from targets are permitted in the short term, allocations are poorly aligned with targets in the long term. Physically feasible allocations differ from logically prescribed ones by 10% to 15%. Moreover, some FPGA slots must remain unallocated. In contrast, when deviations are permitted, STFS exploits this flexibility by tracking history and compensating applications across time for any short-term deviations from the target allocation. Long-term allocations approach their targets, deviations fall to zero, and slot utilization increases by up to 13%.

**Effects of Accelerator Size.** The policy's perspective on fairness determines its sensitivity to accelerator size. STFS allocates fairly with respect to target allocations, ensuring each application receives the targeted number of slots across time. This fairness concept is robust to differences in applications' demands for FPGA slots; larger demands simply lead to less frequent allocations across time. In contrast, round-robin variants allocate fairly with respect to turns on the FPGA, ensuring each application dequeues and computes on the FPGA at regular intervals. Applications that demand more slots will receive more slots across time and strategic users might be incentivized to design larger accelerators. We detail these effects by examining how FPGA slots are divided between accelerators with varied FPGA demands.

Figure 8(a) shows that STFS allocates resources less frequently to an accelerator as its resource demands increase. VITERBI, the largest, demands 5 slots and receives resources in 3 of 10 intervals. AES, the smallest, demands 1 slot and receives resources in every interval. Smaller accelerators, like AES and GSM, can occasionally map multiple instances of their accelerator to the FPGA. In interval 3, GSM receives 4 slots for 2 instances of its accelerator.

Figures 8(b)-8(d) present corresponding round-robin allocations. PRR and RRR favor larger accelerators because they

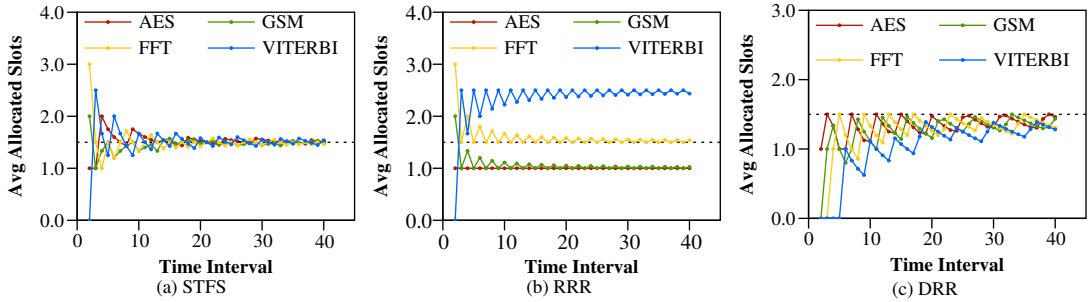


Fig. 4: Average slot allocations on FPGA with six slots. Policies schedule *micro-workload*. (a) STFS, (b) Relaxed Round-Robin, (c) Deficit Round-Robin. Demand for one accelerator instance of {AES, GSM, FFT, VITERBI} is {1, 2, 3, 5}. Target allocation is 1.5 slots per application.

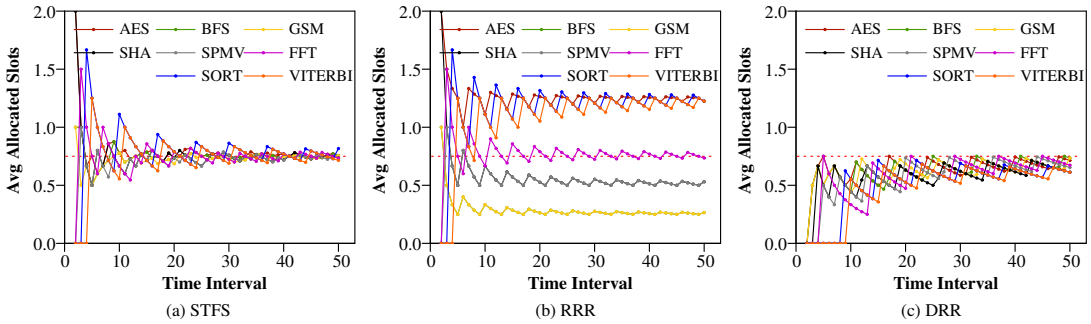


Fig. 5: Average slot allocations on FPGA with six slots. Policies schedule *full-workload*. (a) STFS, (b) Relaxed Round-Robin, (c) Deficit Round-Robin. Demand for each accelerator is detailed in Table II. Target allocation is 0.75 slots per application.

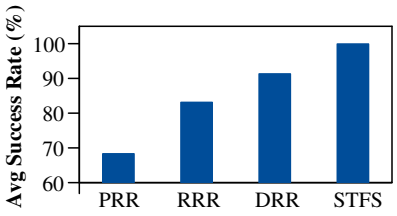


Fig. 6: Average success rate across benchmarks in *full-workload* on FPGA with six slots.

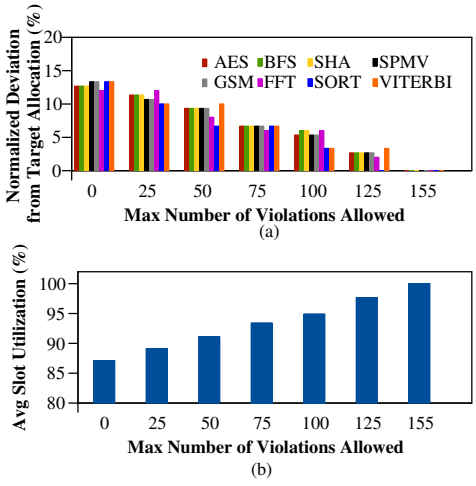


Fig. 7: (a) Deviations from long-term target allocations and (b) Slot utilization as constraints on short-term target allocations are relaxed. STFS schedules *full-workload* on FPGA with six slots across 200 time intervals.

allocate accelerator instances evenly across time regardless of their resource demands. Although VITERBI requires 2.5× more slots than GSM, both receive allocations in the same number of intervals. As a result, large accelerators receive the majority of slots across time. However, PRR and RRR treat smaller accelerators differently. PRR allocates resources to smaller accelerators only when their turn arrives whereas RRR allocates whenever idle resources are insufficient for the next application in round-robin order. Thus, RRR offers more frequent allocations to applications with the smallest demands, such as AES, and achieves greater FPGA utilization when compared to PRR. With DRR, large accelerators, such as VITERBI, are granted resources less frequently because their deficit counters increment at the same rate as all others but decrement at a faster rate due to the larger number of requested and allocated slots. Small accelerators, such as AES, have fewer opportunities to exploit idle FPGA resources because they must hold a sufficiently large counter value to receive an allocation. Even when no other accelerators can use idle slots (*e.g.*, intervals 3, 6, 10), AES lacks the counters needed to receive slots. Thus, the deficit counter restricts allocations and limits utilization.

### *B. Performance*

We evaluate system performance in terms of task throughput, which depends on an accelerator's runtime and size. Figure 9 compares throughput from STFS relative to that from round-robin baselines. One might hypothesize a trade-off between system performance and allocations that align

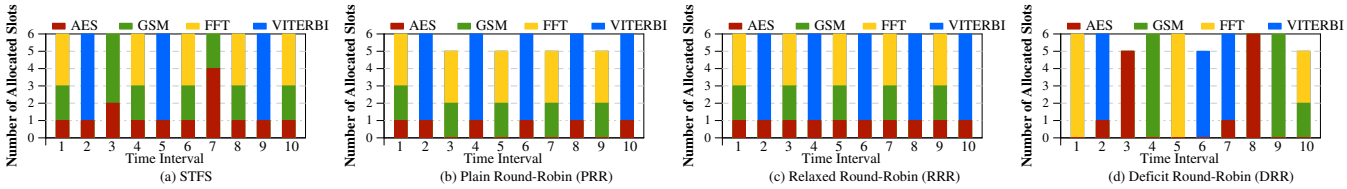


Fig. 8: Slot allocations for *micro-workload* on an FPGA with six slots and ten time intervals. Demand for one accelerator instance of {AES, GSM, FFT, VITERBI} is {1, 2, 3, 5} slots.

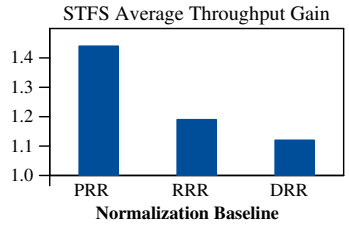


Fig. 9: Benchmarks' throughput from *full-workload* are normalized to those from baseline policies and averaged across all 8 benchmarks on FPGA with six slots.

| | AES | BFS | SHA | SPMV | GSM | FFT | SORT | VITERBI | AVG |
|---|---|---|---|---|---|---|---|---|---|
| S=3 | 1 | 1.97 | 1 | 1.33 | 1 | 1.25 | 1 | 1 | 1.15 |
| S=12 | 0.53 | 0.51 | 0.51 | 1.05 | 1.05 | 0.75 | 0.75 | 0.75 | 0.71 |

TABLE IV: Application throughput on FPGA with 3 and 12 slots relative to that of 6 slots using STFS.

with targets, especially if targets are rooted in some fairness property. But not only does STFS most successfully align allocations with targets, it also outperforms PRR, RRR, and DRR by 44%, 19%, and 12%, respectively.

Table IV examines the impact of slot size on these performance results. For STFS, it reports each application's throughput for FPGAs with 12 slots and 3 slots relative to that with 6 slots. The smaller slots lead to greater intra-slot utilization, which improves task throughput for applications by up to 1.97×. Larger throughput gains are observed for applications that waste more resources when using the coarse-grained slots (*i.e.*, BFS, SPMV, FFT). STFS's average throughput gain across applications with 6 slots is 29% higher than STFS with 3 slots. STFS's throughput gain with 12 slots is 15% higher than its throughput gain with 6 slots.

### C. Resource Utilization

Inter-slot utilization refers to how slots are used across time. In Figure 8, utilization differs across policies as some slots are occasionally left unallocated. STFS's utilization is up to 66% greater than DRR's when comparing interval by interval. STFS's average utilization across time is 13%, 11%, and 16% greater than DRR's when the FPGA is configured for 3, 6, 12 slots, respectively. STFS's average utilization is also 17-23% greater than PRR's, which suffers from rigidly following round-robin order. By tracking history and allocating resources flexibly across time, STFS tolerates short-term deviations from logical shares in exchange for resource efficiency.

Intra-slot utilization refers to how reconfigurable elements—LUTs, FFs, DSP and BRAMs—are used within each slot and is affected by interactions between accelerator demands for slots, slot size, and slot orientation. When the FPGA uses many small slots (*e.g.*, 12), sharing is fine-grained and intra-slot utilization is high. When the FPGA uses a few large slots (*e.g.*, 3), small accelerators are allocated many resources that are left idle. Using small slots improves intra-slot utilization by up to 31%.

### D. Dynamic Workloads

Applications are typically known in advance such that corresponding bitstreams have been generated and compiled in advance of their arrival. However, the run-time system must adjust allocations in response to evolving workload mixes and demands. STFS adjusts allocations upon application arrival or departure. Figure 10 considers the arrival of SPMV and SORT task streams, which join the *micro-workload* workload at runtime. In the left, arrivals occur after STFS has converged to its target allocations. In the right, arrivals occur before STFS has converged and the effects of temporarily deviating from allocation targets have not yet been fully compensated.

STFS responds to arrivals by updating targets and success rates. Supposing targets are set by max-min fairness, additional accelerators reduce target allocations from $6/4 = 1.5$ slots to $6/6 = 1$ slot. Success rates for arriving accelerators are initialized to the largest rate (*i.e.*, lowest priority) of accelerators already in the system, which has two effects. First, STFS preserves relative priorities for existing accelerators, which were set by allocations in previous intervals. Second, arriving accelerators catch up quickly and receive allocations.

Figure 11 considers the departure of the VITERBI task stream. STFS responds by removing the accelerator from request queues, updating targets for remaining accelerators, and allocating slots with success rates calculated from previous intervals. The target increases from $6/4 = 1.5$ slots to $6/3 = 2$ slots after departure. From these results, we conclude that STFS is able to quickly adapt to arrival and departures.

## VI. Related Work

**FPGA Primitives.** FPGAs have been virtualized using overlays [56], [57], [58], [59], [60], [61], [62], spatial regions [63], task virtualization [64], and context switches [65]. Partitioning resources into slots permit concurrent tasks and improves utilization [2], [4], [1], [3], [66], [67], [68], [69]. Slots provide mechanisms for sharing but not policies for run-time management.

Temporal sharing requires dynamic reconfiguration [70], [71], which is difficult due to context switch costs and complex preemptive scheduling [2], [5], [72], [73]. Current tools do not

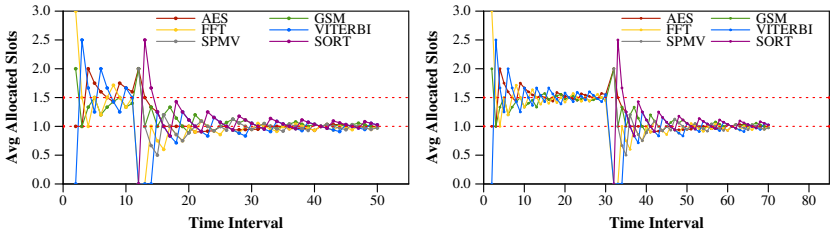


Fig. 10: Average slot allocations on FPGA with six slots. SPMV and SORT join *micro-workload* before (left) and after (right) the system's allocations have converged to target allocations.

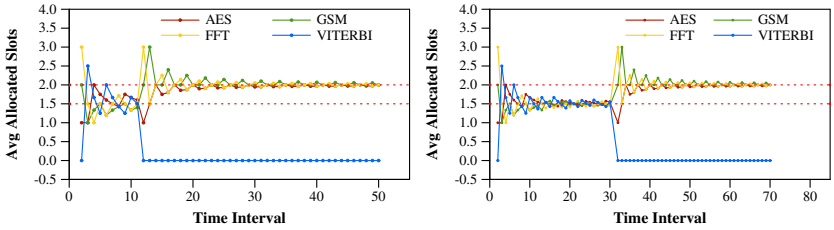


Fig. 11: Average slot allocations on FPGA with six slots. VITERBI departs from *micro-workload* before (left) and after (right) the system's allocations have converged to target allocations.

support automatic preemption and determining reconfiguration points for preemption is difficult. Researchers have explored resource elasticity [9], [20] but, unlike STFS, do not provide schedulers that manage resources according to targets.

**FPGA OS.** Operating systems provide infrastructure and abstractions for virtualization, communication, and management [74], [75], [76], [77], [78], [79]. AmorphOS [5] dynamically scales hardware modules and switches between spatial and temporal sharing. ViTAL [30] provides abstractions for multi-FPGA acceleration, using bitstream relocation on identical, small slots [30]. Recent work propose virtualization for heterogeneous, multi-FPGA systems [80]. STFS's spatiotemporal policies are orthogonal and contribute new management strategies that could be integrated into these frameworks.

**FPGA Scheduling.** A rich body of literature has examined FPGA scheduling. FCFS policies assign resources as tasks arrive [51], but outcomes can be sensitive to arrival order. Greedy policies assign tasks to the smallest slot with sufficient resources [3], [81], but such matching may lead to long wait times and idle resources. Some tasks may be favored during scheduling using priority queues [39] and priorities could be associated with expected acceleration speedups [82]. However, priorities could conflict with notions of fairness and allocation targets, which are the focus on our paper.

Fairness in FPGA management has focused on specific system settings. OPTIMUS schedules tasks with round-robin to ensure equal time on instantiated FPGA accelerators [20]. OPTIMUS supports temporal scheduling for statically configured FPGAs whereas STFS supports spatiotemporal scheduling and allocation for dynamically reconfigured, virtualized FPGAs. This spatial dimension allows STFS to reconcile applications' resource demands and the system's allocation targets. Max-min fairness has been studied for the FPGA's NIC and DMA bandwidth [83]. Vaishnav et al. use partial reconfiguration to dynamically select bigger or smaller accelerators for a task [9], [23] and pursue fairness in space. In contrast, STFS pursues fairness or target allocations in space and time, which is imperative given how FPGA characteristics can lead to a large gap between a policy's logically fair allocation and an FPGA's physically feasible one.

**Non-FPGA Scheduling.** Traditional resources do not present the challenges presented by FPGAs. Offline design exploration yields efficient colocations for core and cache configurations [32], but do not accommodate indivisible resources like FPGA slots. Prior work in distributed shared storage places tenants' partitions on each node's available storage [33], but its fair division assumes a unified pool of resources and neglects cases where demand exceeds capacity. In contrast, FPGA resources are bound to particular slots and applications' demands for slots often exceed capacity.

## VII. Conclusion

STFS is a novel allocation policy that fairly shares the reconfigurable resources of an FPGA between multiple accelerators, via spatial and temporal sharing. Despite the physical constraints on shared FPGAs, which lead to infeasible or inefficient allocations using traditional allocation policies, STFS achieves long-term allocation targets. By permitting short-term deviations from target allocations and compensating those over time, STFS's provides better resource utilization and performance than round-robin baselines.

## Acknowledgment

We thank Anuj Vaishnav and Andrew Putnam for providing guidance on FPGA tools and technologies. We also thank the Xilinx University Program for donating FPGA boards and software. This material is based on work supported by the National Science Foundation under grants CCF-200-2737 and CCF-213-3160.

## REFERENCES


[1] S. Byma, G. Steffan, H. Bannazadeh, A. Garcia, and P. Chow, "FPGA in the cloud: Booting virtualized hardware accelerators with openstack," in *22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014.

[2] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[3] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Proceesings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 430–435.

[4] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *Proceedings of the 15th International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015.

[5] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AMORPHOS," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[6] W. Lie and W. Feng-Yan, "Dynamic partial reconfiguration in FPGA," in *Proceedings of the 3rd IEEE International Symposium on Intelligent Information Technology Application*, vol. 2, 2009, pp. 445–448.

[7] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–39, 2018.

[8] E. J. McDonald, "Runtime FPGA partial reconfiguration," in *Proceedings of the IEEE Aerospace Conference*, 2008, pp. 1–7.

[9] A. Vaishnav, K. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for FPGAs using OpenCL," in *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[10] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *International Workshop on Field Programmable Logic and Applications*. Springer, 2000, pp. 121–130.

[11] M. Happe, A. Traber, and A. Keller, "Preemptive hardware multitasking in ReconOS," in *Proceedings of the International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 79–90.

[12] A. Bourge, O. Muller, and F. Rousseau, "Generating efficient context-switch capable circuits through autonomous design flow," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 1, pp. 1–23, 2016.

[13] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/.

[14] "Deep dive into Alibaba cloud F3 FPGA as a service instances-Alibaba cloud community," https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057.

[15] "Deploy ML models to FPGAs - Azure machine learning — Microsoft docs," https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[16] "Microsoft unveils project Brainwave for real-time AI - Microsoft Research," https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[17] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, 2015.

[18] "Intel® programmable acceleration card with Intel® Arria® 10 GX FPGA," https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac/arria-10-gx.html.

[19] A. Jahanshahi, R. Sharifi, M. Rezvani, and H. Zamani, "Inf4edge: Automatic resource-aware generation of energy-efficient cnn inference accelerator for edge embedded fpgas," in *Proceedings of the 12th IEEE International Green and Sustainable Computing Conference (IGSC)*, 2021, pp. 1–8.

[20] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory FPGA platforms," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 827–844.

[21] "Vivado design suite user guide partial reconfiguration," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug947-vivado-partial-reconfiguration-tutorial.pdf.

[22] C. Kao, "Benefits of partial reconfiguration," *Xcell journal*, vol. 55, pp. 65–67, 2005.

[23] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 4, pp. 1–28, 2020.

[24] "Platform overview — Xilinx runtime 2019.1 documentation," https://xilinx.github.io/XRT/2019.1/html/platforms.html.

[25] K. D. Pham, K. Paraskevas, A. Vaishnav, A. Attwood, M. Vesper, and D. Koch, "ZUCL 2.0: Virtualised memory and communication for zynq ultrascale+ FPGAs," in *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*. VDE, 2019, pp. 1–9.

[26] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A ZYNQ UltraScale+ framework for OpenCL HLS applications," in *5th International Workshop on FPGA for Software Programmers (FSP)*, 2018.

[27] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[28] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, "Prospector: synthesizing efficient accelerators via statistical learning," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 151–156.

[29] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 271–286.

[30] Y. Zha and J. Li, "Virtualizing FPGAs in the cloud," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[31] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–24, 2011.

[32] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 650–664.

[33] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 349–362.

[34] S. M. Zahedi and B. C. Lee, "REF: Resource elasticity fairness with sharing incentives for multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 145–160, 2014.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair allocation of multiple resource types." in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[36] D. C. Parkes, A. D. Procaccia, and N. Shah, "Beyond dominant resource fairness: Extensions, limitations, and indivisibilities," *ACM Transactions on Economics and Computation (TEAC)*, vol. 3, no. 1, pp. 1–22, 2015.

[37] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[38] A. Vaishnav, K. Pham, and D. Koch, "Live migration for OpenCL FPGA accelerators," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 38–45.

[39] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[40] A. Morales-Villanueva and A. Gordon-Ross, "Partial region and bitstream cost models for hardware multitasking on partially reconfigurable FPGAs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 90–96.

[41] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2007, pp. 188–196.

[42] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 65–74, 2009.

[43] S. M. Zahedi, S. Fan, and B. C. Lee, "Managing heterogeneous datacenters with tokens," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 2, pp. 1–23, 2018.

[44] B. Gottschall, T. Preußer, and A. Kumar, "Reloc—an open-source Vivado workflow for generating relocatable end-user configuration tiles," in *Proceedings of the 26th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[45] S. Dhar, S. Adya, L. Singhal, M. A. Iyer, and D. Z. Pan, "Detailed placement for modern FPGAs using 2d dynamic programming," in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, pp. 1–8.

[46] S. Dhar, M. A. Iyer, S. Adya, L. Singhal, N. Rubanov, and D. Z. Pan, "An effective timing-driven detailed placement algorithm for FPGAs," in *Proceedings of the ACM International Symposium on Physical Design*, 2017, pp. 151–157.

[47] S. Dhar and D. Z. Pan, "GDP: GPU accelerated detailed placement," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[48] J. Landgraf, T. Yang, W. Lin, C. J. Rossbach, and E. Schkufza, "Compiler-driven FPGA virtualization with synergy," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 818–831.

[49] "Petalinux tools," https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html#tools.

[50] "Using Xilinx SDK," https://www.xilinx.com/htmldocs/xilinx2017_1/SDK_Doc/index.html.

[51] M. Huang, D. Wu, C. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale," in *Proceedings of the ACM Symposium on Cloud Computing*, 2016.

[52] B. Reagen, R. Adolf, Y. Shao, G. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2014.

[53] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1995, pp. 231–242.

[54] J. Khamse-Ashari, G. Kesidis, I. Lambadaris, B. Urgaonkar, and Y. Zhao, "Max-min fair scheduling of variable-length packet-flows to multiple servers by deficit round-robin," in *Proceedings of the Annual IEEE Conference on Information Science and Systems (CISS)*, 2016.

[55] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[56] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Proceedings of the 20th IEEE International Symposium on Field-programmable Custom Computing Machines*, 2012, pp. 93–96.

[57] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *2013 23rd IEEE International Conference on Field programmable Logic and Applications*, 2013.

[58] D. Koch, C. Beckhoff, and G. G. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," in *Proceedings of the 23rd IEEE International Conference on Field Programmable Logic and Applications*, 2013, pp. 1–8.

[59] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on DSP blocks," in *Proceedings of the 23rd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 25–28.

[60] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[61] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on FPGAs?" in *Proceedings of DASC/PiCom/DataCom/CyberSciTech*. IEEE, 2016.

[62] C. Liu, H.-C. Ng, and H. K.-H. So, "QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*. IEEE, 2015, pp. 56–63.

[63] K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell, "Microkernel hypervisor for a hybrid ARM-FPGA platform," in *Proceedings of the 24th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2013, pp. 219–226.

[64] C. Plessl and M. Platzner, "Zippy-a coarse-grained reconfigurable array with support for hardware virtualization," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2005, pp. 213–218.

[65] T. Y. Lee, C. C. Hu, L. W. Lai, and C. C. Tsai, "Hardware context-switch methodology for dynamically partially reconfigurable systems," *Journal of Information Science and Engineering*, vol. 26, no. 4, 2010.

[66] O. Knodel and R. Spallek, "RC3E: provision and management of reconfigurable hardware accelerators in a cloud environment," *arXiv preprint arXiv:1508.06843*, 2015.

[67] O. Knodel, P. R. Genssler, and R. G. Spallek, "Virtualizing reconfigurable hardware to provide scalability in cloud architectures," in *Proceedings of the International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, 2017.

[68] O. Knodel and R. G. Spallek, "Computing framework for dynamic integration of reconfigurable resources in a cloud," in *2015 Euromicro Conference on Digital System Design*. IEEE, 2015, pp. 337–344.

[69] K. Manev, A. Vaishnav, and D. Koch, "Unexpected diversity: Quantitative memory analysis for zynq ultrascale+ systems," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 179–187.

[70] W. Wang, M. Bolic, and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2013, pp. 1–9.

[71] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "Automatic virtualization of accelerators," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 58–65.

[72] M. Paolino, S. Pinneterre, and D. Raho, "FPGA virtualization with accelerators overcommitment for network function virtualization," in *Proceedings of the IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–6.

[73] S. Pinneterre, S. Chiotakis, M. Paolino, and D. Raho, "vFPGAmanager: A virtualization framework for orchestrated FPGA accelerator sharing in 5g cloud environments," in *Proceedings of the IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, 2018.

[74] H. K. H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using borph," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, pp. 1–28, 2008.

[75] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, pp. 1–33, 2009.

[76] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, 2005, pp. 8–pp.

[77] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proceedings of the 19th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 170–177.

[78] K. Fleming and M. Adler, "The leap FPGA operating system," in *FPGAs for Software Programmers*. Springer, 2016, pp. 245–258.

[79] 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 *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017.

[80] Y. Zha and J. Li, "When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 123–134.

[81] H. L. Kidane, E.-B. Bourennane, and G. Ochoa-Ruiz, "NoC based virtualized accelerators for cloud computing," in *Proceedings of the 10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, 2016, pp. 133–137.

[82] G. Dai, Y. Shan, F. Chen, Y. Wang, K. Wang, and H. Yang, "Online scheduling for FPGA computation in the cloud," in *Proceedings of*

*the International Conference on Field-Programmable Technology (FPT)*, 2014.

[83] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen, "Fpga resource pooling in cloud computing," *IEEE Transactions on Cloud Computing*, vol. 9, no. 2, pp. 610–626, 2018.