Nimblock: Scheduling for Fine-grained FPGA Sharing through Virtualization

Meghna Mandava
meghnam4@illinois.edu
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA

Paul Reckamp
paulrr2@illinois.edu
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA

Deming Chen
dchen@illinois.edu
University of Illinois, Urbana-Champaign
Urbana, Illinois, USA

ABSTRACT

As FPGAs become ubiquitous compute platforms, existing research has focused on enabling virtualization features to facilitate fine-grained FPGA sharing. We employ an overlay architecture which enables arbitrary, independent user logic to share portions of a single FPGA by dividing the FPGA into independently reconfigurable slots. We then explore scheduling possibilities to effectively time- and space-multiplex the virtualized FPGA by introducing Nimblock. The Nimblock scheduling algorithm balances application priorities and performance degradation to improve response time and reduce deadline violations. Unlike other algorithms, Nimblock explores preemption as a scheduling parameter to dynamically change resource allocations, and automatically allocates resources to enable suitable parallelism for an application without additional user input. In our exploration, we evaluate five scheduling algorithms: a baseline, three existing algorithms, and our novel Nimblock algorithm. We demonstrate system feasibility by realizing the complete system on a Xilinx ZCU106 FPGA and evaluating on a set of real-world benchmarks. In our results, we achieve up to 5.7× lower average response times when compared to a no-sharing and no-virtualization scheduling algorithm and up to 2.1× average response time improvement over competitive scheduling algorithms that support sharing within our virtualization environment. We additionally demonstrate up to 49% fewer deadline violations and up to 2.6× lower tail response times when compared to other high-performance algorithms.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Reconfigurable computing**.

KEYWORDS

Virtualization, Reconfigurable Computing, Real-time Scheduling, Overlay Architectures

ACM Reference Format:
Meghna Mandava, Paul Reckamp, and Deming Chen. 2023. Nimblock: Scheduling for Fine-grained FPGA Sharing through Virtualization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589095

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589095

1 INTRODUCTION

Field-programmable gate arrays (FPGAs) are quickly becoming commonplace in both edge and data-center computing, enabling acceleration of wide classes of applications [4, 7, 8, 17, 22, 31] more flexibly, at lower power and lower latency than CPUs and GPUs [2, 18, 32]. The flexibility of FPGAs enables users to design application specific solutions in an efficient and cost-effective manner while meeting desired quality of service (QoS) standards. Thus, FPGAs are a coveted resource in commercial cloud platforms such as Amazon AWS [1] or Microsoft Azure [25].

CPU compute resources have traditionally been virtualized with virtual machines or an operating system, allowing multiple programs or users to run on the CPU simultaneously. This is accomplished through space-multiplexing, by assigning a subset of cores to a user, or time-multiplexing, by assigning time-slots on cores to an application. Unlike CPUs, FPGAs are generally programmed with a single bitstream requiring independent users to cooperate to program the device [19] or wait their turn in line. Traditionally, coarse-grained FPGA sharing meets the demand for FPGA compute resources in the cloud by implementing queues to time-multiplex a hardware resource, allowing high-priority applications to bypass or jump the queue. However, time-multiplexing of FPGAs is blocked by the large overhead of a context switch, involving a long reconfiguration of the programmable logic in addition to data management. Moreover, the FPGAs are typically space-multiplexed at the single FPGA granularity, dedicating entire pieces of hardware to a single job, regardless of whether or not the job needs to use all the resources on the FPGA, potentially leading to resource under-utilization. Because of these multiplexing issues, resource requirements are satisfied by increasing the number of FPGAs and implementing queues—AWS EC2 F1 instances [1], for example, provide FPGAs in totality to the end user. FPGAs have also been used to accelerate data-center services which can require the use of multiple devices. Previous solutions have used large-scale fabrics, allocating entire FPGAs or multiple FPGAs to tasks [28]. However, many workloads may not require using the FPGA for extended periods of time or the entire FPGA; here, fine-grained sharing can provide flexibility and resource efficiency.

FPGA virtualization strives to bring virtualized features to FPGAs in order to use resources more efficiently by designing overlays, tool-flows, and hypervisors. Notably, a virtualized FPGA would support:

(1) Fine-grained multi-tenancy, allowing multiple independent applications to share the FPGA simultaneously.
(2) Scale-out, allowing applications to spread across multiple FPGAs.
(3) Operating system-like features such as virtual memory support or a networking stack.

Combined, these features present the illusion of an infinite, homogeneous, and reconfigurable fabric to the end user. FPGA virtualization can allow for efficient sharing of FPGAs in cloud and edge computing and ease the burden on the developer.

Recent works explore using FPGAs in a serverless computing model [9, 13, 30] without virtualizing the FPGAs. Heterogeneous serverless computing has been explored with GPUs, enabled by GPU virtualization [20, 26]. Ultimately, FPGA-supported serverless computing would need to rely on virtualizing FPGAs as well, to improve utilization efficiency and facilitate flexible resource allocation. It is also necessary for cloud services such as Function-as-a-Service (FaaS), which require strong isolation between applications. This could be accomplished using OS-like abstractions to isolate applications on an FPGA as would be done on a CPU [21]. Combined with proper task scheduling and resource allocation, FPGA virtualization will become an essential enabler for serverless computing with FPGAs as a first-class citizen providing customized compute acceleration.

Typical solutions for FPGA virtualization [29, 35] create a custom FPGA overlay that splits the fabric into *slots*, which are independently reconfigurable blocks (or tiles) that are able to host arbitrary logic in the system. We define an overlay as the hardware/software layer which is placed on top of the FPGA's physical resources. Surrounding the reconfigurable slots is the *static region* of the overlay which facilitates coordination and slot management. The reconfigurable nature of an FPGA supports fine-grained sharing in the virtualized system through a technique known as dynamic partial reconfiguration (DPR). DPR enables portions of the fabric to be reconfigured independently during runtime and in a fraction of the time of regular reconfiguration [33].

Once infrastructure for FPGA virtualization is in place, a complementary scheduling problem must be solved to efficiently extract performance. The hypervisor must select from a pool of applications to determine start times and select slot allocations to determine application placement. Further complicating the scheduling problem, the system will be limited by the reconfiguration overhead and the inability to configure more than one tile simultaneously on a single device.

Searching for an optimal scheduling solution in this space is challenging and often infeasible under real-time constraints, so heuristic scheduling approaches are often taken [3, 21, 24, 34]. Moreover, the heuristic scheduler should make practical considerations for FPGA sharing. First, in some sharing contexts, applications are not known ahead of time and arrive in unpredictable intervals. Second, in a data-center or real-time system, applications often have different priorities or deadlines to meet. Finally, the scheduler should consider performance optimizations within an application. Many cloud applications are deployed in batches to improve efficiency. Pipelining across batches to hide reconfiguration time behind active computation is one such optimization. In FaaS or serverless platforms, the scheduler must consider allocating resources for individual tasks, rather than entire applications. Therefore, support for fine-grained scheduling is important for such computing models in the cloud. At its core, effective FPGA sharing for virtualization must be able to schedule arbitrary applications in an efficient manner while hiding the reconfiguration latency and managing priority levels and deadlines.

In this work, we present Nimblock, an exploration of fine-grained sharing techniques on FPGAs with consideration for arbitrary real-time workloads and priority levels. Nimblock employs an overlay architecture with reconfigurable slots to split an FPGA into independent virtual tiles. The Nimblock runtime can extract performance by sharing the FPGA among applications that arrive in real-time with varying batch sizes and priority levels. A space-multiplexing overlay with virtualization can improve performance of FPGAs in the cloud and Nimblock's advanced, yet efficient, scheduler further improves performance of applications with such an overlay. We summarize our contributions as follows:

(1) We present a novel scheduling algorithm that allows both time- and space-sharing of multiple slots on an FPGA and considers applications with arbitrary arrival times and priorities. The algorithm enables pipelining of tasks belonging to the same application across different batches. Unlike previous space-sharing implementations, Nimblock also enables preemption of tasks. We evaluate our algorithm on a ZCU106 FPGA board and validate its efficacy by measuring results from real workloads running on the board with different real-time constraints and priority levels.
(2) We implement a hypervisor on top of a time- and space-sharing overlay that supports DPR. The hypervisor, which runs the Nimblock scheduling algorithm, allocates resources and schedules tasks dynamically, making it well-suited for real-time scheduling.
(3) We demonstrate up to a 5.7× average response time improvement over a baseline scheduling approach that does not support sharing and virtualization and up to 2.1× average response time improvement over other high-performance scheduling algorithms.
(4) We show up to a 2.6× reduction in tail response time compared to other high-performance algorithms. We also achieve up to a 49% lower deadline violation rate than previous advanced scheduling algorithms.
(5) We open-source our FPGA virtualization system, including the hypervisor software and the FPGA design, benefiting the computing industry and the research community.

The rest of this work is outlined as follows. In Section 2 we describe the Nimblock system implementation. We then motivate the Nimblock scheduling algorithm in Section 3 and present the Nimblock scheduling algorithm in Section 4. We discuss our evaluation methodology and results in Section 5 before surveying related work in Section 6 and concluding in Section 7.

2 NIMBLOCK SYSTEM

2.1 Overlay

The Nimblock overlay, inspired by the overlay used in [12], consists of two components that enable fine-grained sharing and DPR on

the target FPGA—a software element and an FPGA tiling scheme. Depending on the target device, the software element can run on the processing system (PS) or other embedded processor. In FPGA devices without embedded CPUs, the host CPU would manage communication and control over a PCIe interface. A simplified view of the overlay is given in Figure 1.

At the core of the PS portion of the overlay is the embedded ARM core, which runs the Nimblock hypervisor. It also manages accelerator data, application bitstreams, and reconfiguration. Partial bitstreams for each slot are stored on the SD card and loaded into memory by the ARM core on demand. The ARM core can access the partial bitstream and send a request to the configuration access port (CAP) which reconfigures the defined portions of the FPGA. Reconfiguration speed is constrained by the internal bandwidth of the CAP interface (dictated by the device) and the size of the reconfigurable portion (dictated by the overlay structure). Communication between the PS and FPGA is handled by memory-mapped interfaces enabling control registers to be written by the PS and reconfigurable regions to access the shared system memory.

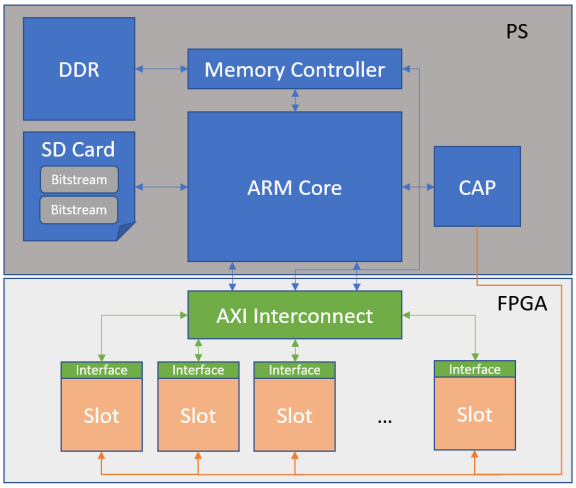


Figure 1: Nimblock overlay. Static elements are in green, reconfigurable elements are in orange.

The FPGA portion of the Nimblock overlay is split into a static region, programmed once at system start-up, and multiple reconfigurable slots which are programmed dynamically by user logic. The static region consists of interconnects which connect the slots to the PS and the system memory as well as decoupling resources to isolate configurable logic during reconfiguration. On our system, inter-slot communication is performed through the PS. It is possible to improve this architecture with different methods for inter-slot communication, such as a Network-on-Chip (NoC). We plan to explore this optimization in future work.

The reconfigurable slots are floorplanned to be uniform, have the same resource size, and can host arbitrary user logic. The size of the slots is determined by the constraints of the FPGA and the requirements of the applications to be run on the FPGA. A slot must be sufficiently large to allow enough resources for the largest application task which we may map onto the FPGA, but no larger, as it is beneficial to have more slots available. If the typical type of application which users will run on the FPGA is known, it is possible to choose slots such that they best fit the application. The efficiency of resource utilization is determined by how well the tasks of an application are partitioned. Nimblock aims to improve performance regardless of the slot partitioning and application partitioning, and it is flexible across different numbers of slots. For our purposes, user logic is configured to have a single memory-mapped interface for control and second memory-mapped interface for data. The control interface is used by the hypervisor to launch tasks on user logic. The data interface is used by the user logic to read from and write to data buffers. This is easily achievable through high-level synthesis flows using interface pragmas which optimize latency, throughput, and area.

2.2 Hypervisor

The Nimblock hypervisor runs on the embedded ARM core. As system manager, the hypervisor drives reconfiguration, manages application data, and runs the scheduling algorithm outlined in Section 4. The process of preparing an application and adding it to the hypervisor proceeds as follows.

Before sending a request to the hypervisor, the application is partitioned into slot sized *tasks*—each task is a portion of the application with an input and an output. In turn, these tasks are composed into a *task-graph*, a Directed Acyclic Graph (DAG) with nodes representing tasks and edges representing dependencies. For example, if the application is LeNet, its six layers could be split into three tasks: The first task is comprised of the first convolution and pooling layers, the second is the next convolution and pooling layers, and the third is the last convolution layer and the fully connected layer. Each node in the graph would be a grouping of two layers with two edges linking the three nodes in a chain. An automatic flow may be used to appropriately partition applications into tasks which fit the slot [14, 15, 35]. Ideally, the tasks would be generated through partitioning such that the user logic uses as much of the slot as possible. The application can also be manually partitioned in a similar manner. As previously mentioned, the Nimblock compilation flow is agnostic to slot sizes. Similarly, it would function with either automatic task partitioning or with manual partitioning.

The partitioned application is then placed in a partial reconfiguration flow to generate partial bitstreams for each of the tasks in the application. For the purpose of this work, the flow generates a partial bitstream for each task in an application which can map to each slot in the overlay, i.e., for n slots on the FPGA, each task will have n partial bitstreams, to provide complete flexibility for task scheduling. This bitstream generation is done using a script and is automated. Partial bitstream relocation can reduce the number of bitstreams we need to store and generate, but exploration of this concept [5, 10, 23] is beyond the scope of this work. The bitstreams are added to the hypervisor with a header that provides interface information, application batch size, HLS performance estimates, and priority level. The interface information and performance estimates are parsed from the HLS output, while the batch size and priority level are user specified parameters. Our testbed compiles this information as part of C header files, but a deployed system could easily parse the information from a JSON file. For the purpose

of this work, we define batch size as the number of independent application inputs requested to be executed by a single user at once. Note this definition is the same as the one used in [12].

When the bitstreams arrive at the hypervisor, they are placed in the filesystem (the SD card in our system) and the pending application is added to an application queue to wait for scheduling. When a task of the application is selected by the scheduler, the bitstream is loaded from the SD card to system memory and the hypervisor requests a reconfiguration through the CAP APIs on the system. After reconfiguration is complete, the hypervisor allocates buffers and launches the task. Tasks read inputs and write outputs to and from the allocated buffers. Tasks which use data computed by a preceding task will obtain the data from buffers allocated by the hypervisor. When the task is complete, the hypervisor relinquishes the unneeded data buffers and marks the slot as open for use. When all tasks in an application are complete, the hypervisor sends a response with the result data.

3 MOTIVATION

3.1 Scheduler Goals

The primary metric to measure system performance in a real-time sharing scenario is the response time of an application. The response time of application i, T_i, is defined as the difference between the application's arrival time, A_i, and the retirement time, R_i. This includes the time to finish all batches of the application. As a general objective, our scheduling algorithm should seek to minimize the average response time of the N pending applications.

Equally important is the scheduling algorithm's ability to meet application QoS guarantees; a system which makes QoS guarantees for high-priority applications should uphold its promise even under heavy loads and resource contention. Therefore, our co-objective is to reduce the number of deadline violations and demonstrate a tighter deadline guarantee than other scheduling algorithms. These two goals inform the design of our scheduling algorithm and are impacted by the ways that we share and schedule on the FPGA.

3.2 Sharing Modes

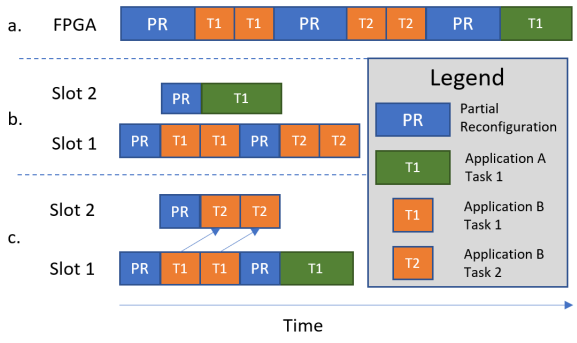


Figure 2: Sharing modes for multiple FPGA slots. Arrows represent dependencies across pipeline stages.

In a naive system with only temporal multiplexing, an application's tasks are simply scheduled in sequential order as they arrive, requiring little scheduling overhead, but causing serialization as applications arrive to a full system as seen in Figure 2(a). By partitioning the FPGA into slots, task-level parallelism is extracted from independent tasks. This approach overlaps reconfiguration time with computation and enables tasks to execute simultaneously for multiple applications. as seen in Figure 2(b). In the two previous cases, elements of an application's batch are bulk processed—tasks will be executed multiple times between reconfiguration.

A final use for fine-grained sharing is to allow tasks within the same application to pipeline across slots. Many workloads are submitted to accelerators in batches. Kernels are typically run on large datasets, and it is common for accelerators to perform batch processing on multiple inputs in order to minimize human interaction. Schedulers can leverage this in order to efficiently use resources. In the fine-grained sharing mode, an application's tasks can co-exist on the FPGA while working on different batch items. Pipelining across batches enables us to extract additional performance and reduce the response time of applications. For example, when task 1 finishes processing the first input, task 2 can start processing its first input while task 1 processes the second input. This mode enables a single application to further reduce its response time at the cost of slot monopolization as seen in Figure 2(c).

For a single application, we see that large batch sizes allow for slots to be used more effectively than small batches. Large batches are able to hide the latency required for reconfiguration; the reconfiguration time takes up a higher percentage of the overall latency for smaller batch sizes [12]. Once the task pipeline is established, the scheduler avoids performing unnecessary scheduling decisions which would be required if the same application were run in smaller batches. Hence, it is beneficial for applications that are eligible to run with large batch sizes to do so. The Nimblock scheduler will take advantage of this whenever possible, especially because the reconfiguration time can be substantial. However, when an application is pipelined across batches it consumes multiple slots simultaneously. Rampant pipelining can choke performance for applications arriving later, causing resource starvation and deadline violations. For example, if a long-running (whether through application latency or batch size) application with many tasks arrives at the system first and aggressively pipelines across slots to minimize its own response time, it monopolizes resources. Newly arriving applications will be left with no resources available until the long-running application completes.

To prevent this, we need a method to reverse this scheduling decision and roll back the later pipeline stages. We can accomplish rollback by pausing execution of the long-running application's task and configuring the newly arriving application in its place. When additional resources are available, the preempted tasks will be rescheduled and run to completion. By *preempting* applications in this way, we introduce an additional reconfiguration as overhead but are able to reverse scheduler decisions that are obsolete due to new information.

We refer to this method as *batch-preemption*. The decision to add this functionality to Nimblock is not easy. Schedulers must already select tasks to schedule and select slots to schedule them to while considering priorities and waiting times. Preemption can result in performance degradation if implemented poorly and is unpopular among current approaches [21, 34]. Moreover, classic preemption

requires checkpointing arbitrary FPGA state, but checkpointing is challenging—user state includes stateful logic blocks in the FPGA which are difficult to capture [21]. In order to avoid saving user logic state, batch-preemption stops execution at batch breakpoints and does not need to save the user logic state. As such, this method is both efficient and effective. Despite these challenges, the Nimblock scheduling algorithm employs fine-grained sharing, pipelining, and batch-preemption to improve application performance.

4 SCHEDULING ALGORITHM

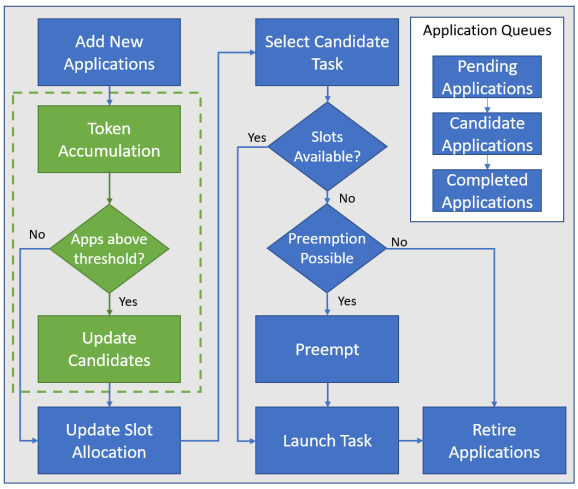


Figure 3: Nimblock scheduling algorithm. Green denotes elements borrowed from PREMA [3, 34].

Figure 3 presents a block diagram showing the major operational steps of our scheduling algorithm. First, all new applications are added to the scheduler and initialized. Second, applications accumulate scheduling tokens, which determine its candidacy to be scheduled to run, and the candidate application pool is updated (Section 4.1). Third, we update the slot allocation for each candidate application (Section 4.2). We then select a task to schedule (Section 4.3) and a slot to reconfigure, preempting existing tasks if necessary (Section 4.4). Lastly, the selected task is launched and we retire any completed applications.

4.1 Candidate Applications

In order to reduce response times, we need to schedule applications that have waited the longest to execute, but we must also ensure that high-priority applications are executed quickly. To balance these factors, we utilize the token accumulation strategy from the PREMA scheduling algorithm [3, 34]. This strategy uses tokens as a mechanism for deciding when to schedule applications to be run. Consistent with previous work, our implementation uses three increasing levels of priority: 1, 3, 9. As applications wait to be scheduled, they accumulate tokens proportional to the application performance degradation and the application priority level. Applications accumulate tokens at set scheduling intervals, when new applications are added, and when an application completes. Existing implementations using PREMA target Deep Neural Network (DNN) computation and use models to estimate performance; we leverage performance estimates from high-level synthesis (HLS) EDA tools. From the HLS output, we obtain a latency estimate for each task. We then analyze the application's task-graph and sum task latency estimates to obtain an application latency estimate.

Algorithm 1 Candidate selection

1: **initialization**
2: **for** application $a \in$ arrival queue A **do**
3: $\quad a.token \leftarrow a.priority$
4: **end for**
5: **for** application $a \in$ pending queue R **do**
6: $\quad a.token \leftarrow a.token + \alpha \times a.priority \times degradation_{norm}$
7: **end for**
8: $threshold \leftarrow \max([floor_{prio}(a.token) \text{ for } a \in R])$
9: $candidates \leftarrow [a \text{ for } a \in R \text{ if } a.token > threshold]$
10: **return** $candidates$

When an application arrives, it moves from the arrival queue to the pending application queue and accumulates tokens as described in Algorithm 1, line 6. The application is initially assigned tokens according to its priority level. As applications accumulate tokens, they increase the candidate threshold using the PREMA thresholding method (Algorithm 1, line 8). The threshold is the maximum token number in the pending application queue rounded down to the nearest priority level from the priority levels in the system. Applications with token numbers greater than the threshold are considered candidates.

4.2 Slot Allocation

The Nimblock scheduling algorithm triggers a *reallocation* at periodic scheduling intervals, along with whenever the candidate application pool changes, where it decides how to allocate slots in a way that reduces response times and ensures deadlines are met. As can be seen in Figure 3, Nimblock's scheduling algorithm starts to differ significantly from PREMA from this step and on. Our slot allocation algorithm is motivated by two observations. First, it is beneficial to ensure that all candidate applications have access to at least one slot to ensure forward progress and prevent additional waiting time from damaging response time numbers. Second, applications that were added to the candidate application pool first have experienced the most performance degradation and should receive resources first. Therefore, we first allocate one slot to each candidate. If there are more candidates than slots in the system, we first allocate slots to the oldest applications.

Depending on the number of candidates, there may be additional slots available for allocation. In order to allocate these slots fairly, we introduce the *goal number*, inspired by an observation in DML [12]. The authors of DML noted that applications intuitively have a limit to the number of slots they can effectively utilize, determined by the maximum level of parallelism that can be extracted from the application. To leverage this, we will use similar analysis to identify the *saturation point* of an application, the point at which allocating additional slots results in no or marginal performance improvements.

To identify the saturation point of an application, we need to generate performance estimates across batch sizes and slot allocations. To accomplish this, we leverage the integer linear programming (ILP) formulation from DML [12] which accounts for pipelining and reconfiguration time. We then generate a task-graph, inserting nodes for partial reconfiguration between compute nodes. The task-graph is transformed into an ILP using Python and solved using Gurobi [16]. We sweep the number of slots from one to the number of slots in the system, and identify the point where adding additional slots provides little performance improvement. Because the ILP solver relies only on early performance estimates of the application, saturation point analysis happens in parallel with synthesis, place and route, and bitstream generation for the tasks, keeping such an analysis firmly off the critical path of the user flow.

We then inspect the saturation points and identify goal numbers for applications. Across all applications, we note that allocating a second slot provides the greatest benefit—a second slot enables multiple batches to execute in parallel for a single application. Applications with additional parallelism in their task-graphs further benefit from slots up to the number of parallel paths in the graph.

If additional slots remain after increasing all candidates' slot allocation to the goal number, we assign additional slots to any application that can make use of them in order of application age. This enables older applications to maximize performance to meet their deadlines by pipelining their tasks.

4.3 Task Selection

Since only one slot can be reconfigured on a single FPGA at a time, Nimblock provides a method to select a task from a candidate application to schedule at each interval. We prioritize the oldest application in the candidate pool to minimize additional performance degradation. Pipelining between batches is begun automatically if an application has slots available to opportunistically take advantage of excess resources. If no task is ready to be scheduled, nothing is done. On the other hand, if there is a task ready to be scheduled, but no available slots, we consider preempting an existing application.

4.4 Preemption

As we saw in Section 4.2, slot allocations are updated as the candidate application pool changes. As new applications arrive, some existing applications will be reassigned fewer slots than they are currently using. As discussed in Section 3.2, applications with large batch sizes may better utilize resources by pipelining tasks. However, without a method to intervene, these applications will become over-consumers and continue to utilize their slots until completion. In the worst case, this could delay the execution of other applications and increase the number of deadlines violated. To avoid resource monopolization, Nimblock introduces batch-preemption for FPGAs, enabling both fine-grained time- and space-multiplexing in our runtime.

If an application is ready to execute and there are no available slots to schedule it to, we begin our batch-preemption algorithm as seen in Algorithm 2. In order to facilitate consistent state checkpointing, we elect to preempt only at batch breakpoints (when a task is waiting to have its next batch launched). This addresses the checkpointing issues for FPGA by requiring us to capture only application state. The algorithm iterates over all running applications and evaluates its ability to be preempted. We select the candidate application that has surpassed its slot allocation by the most for batch-preemption. Algorithm 2, line 4 shows how we calculate an application's over-consumption using the number of slots which have been allocated to the application as the goal, *a.slots_allocated*, and the number of slots currently being used by the application, *a.slots_used*. This is the application that will experience the least performance impact from removing a slot. After selecting the application to preempt, we find the task in that application's task-graph that is latest in topological execution order and select that task to remove; this eliminates the chance of removing a task that is acting as a pipelined dependency for another currently running task. If this task is currently in the middle of executing a batch, we delay preemption. Once the task reaches a batch boundary, i.e., the task has finished executing a batch of data and is currently awaiting the next batch, the batch state of the preempted task is saved (Algorithm 2, line 12). The slot which the preempted task occupied is then reconfigured with the new candidate task. If no application is an over-consumer, then no task will be preempted.

Algorithm 2 Preemption

```
1:  over-consumption ← 0
2:  for slot s ∈ slots do
3:      a ← s.application {assess slot's current application}
4:      consumption ← a.slots_used − a.slots_allocated
5:      if s.task_is_waiting and consumption > over-consumption
        then
6:          over-consumption ← consumption
7:          over-consumer ← a
8:      end if
9:  end for
10: tasks ← topological_sort(over-consumer.running_tasks)
11: preempt_task ← tasks.end()
12: if preempt_task.slot.waiting_for_next_batch then
13:     return preempt_task.slot
14: end if
```

Table 1: Slot and Static Region Utilization

Region	DSP	LUT	FF	Carry	RAMB18	RAMB36	IOBuf
Slot	46-92	9680-12960	19360-22880	1210-1620	44-46	22-23	1908-2343
Static	1004	122560	245120	15320	172	86	24803

5 EVALUATION

5.1 Methodology

We evaluate our scheduling algorithm on a ZCU106 FPGA from Xilinx. The static overlay is partitioned into ten slots. As mentioned in Section 2.1, Nimblock is adaptable to different slot and task configurations. We partition the board based on the resources available on the ZCU106 and the benchmark applications which we intend to allocate on the slots. Table 1 shows the utilization of the regions on the board.

We run our hypervisor as a bare-metal application on the embedded ARM core. To emulate real-time application arrival on a

single FPGA, we create a testbed environment. The testbed reads in a sequence of *events*, where an event is defined as the arrival of an application at the hypervisor and contains an application name, batch information, priority level, and arrival time. The event is released to the hypervisor after the event's arrival time has passed. We select applications from the suite used in a related work [12]: 3D-rendering, digit recognition, optical flow, image compression, LeNet, and AlexNet. The former three are from the Rosetta benchmark suite [38], and the latter three are custom benchmarks. The chosen benchmarks are intended to represent real-world benchmarks from different domains. Although these applications are feed-forward, represented as DAGs as done in previous works, Nimblock is a general solution applicable to applications with different characteristics.

We manually partitioned the benchmarks into optimized, unique tasks such that each one can fit in a single slot on the FPGA and generated bitstreams using Vivado 2019.1. Corresponding task-graphs were also manually constructed with the knowledge of the tasks, and the resulting graphs were provided to the hypervisor with each application. Table 2 provides further details on the partitioning and size of the benchmarks. It shows the number of tasks into which each application is partitioned and the number of edges in the graph. The applications vary in the number of tasks and task-graph complexity, and therefore also vary in their potential for parallelism. An example of a task-graph for a complex benchmark, AlexNet, is shown in Figure 4.

Table 2: Benchmark Sizes

Benchmark	Number of Tasks	Number of Edges
LeNet (LN)	3	2
AlexNet (AN)	38	184
Image Compression (IMGC)	6	5
Optical Flow (OF)	9	8
3D Rendering (3DR)	3	2
Digit Recognition (DR)	3	2

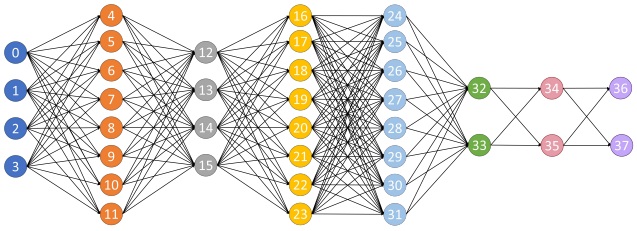


Figure 4: Task-graph for AlexNet. Each vertex represents an individual task. Vertices with the same color represent identical tasks which a layer is split into. Directed edges show task dependencies.

When the events are released to the hypervisor, they are placed into the hypervisor's pending application queue and the hypervisor executes as described in Section 2.2. When an application is completed, the hypervisor stores application metadata until the entire test sequence is completed for result collection.

We evaluate five scheduling algorithms. Our baseline algorithm is a "no-sharing" algorithm where only one application is able to use the entire FPGA at a time; applications wait in a pending application queue until it is their turn to execute. In this scheme, when an application is selected to run, it is able to make use of all slots on the board to execute parallel branches of its task-graph. In addition to the no-sharing baseline algorithms, we implement and compare against three task-based heuristic algorithms.

The first is a naive first-come, first-served (FCFS) scheduling algorithm where all tasks that are ready to execute from all applications are selected in the order that they arrived. Applications are able to execute parallel paths simultaneously, but may not have access to as many resources as in the baseline. On the other hand, applications may see response time reduction due to reduced waiting times.

The second is a task-based PREMA algorithm. Existing implementations of PREMA scheduling algorithms [3, 34] are custom-tuned to their target systems and do not map directly to our sharing scheme. To have a fair comparison, we keep the token accumulation scheme as well as the candidate selection methodology of choosing the shortest candidate to execute next for PREMA. We compare to the PREMA scheme from [23] because it targets a multi-slot system similar to ours. This scheme does not have advanced features such as preemption and pipelining across different batches.

The third algorithm is a queue-based round-robin (RR) scheduling algorithm adapted from the implementation in [21]. Using their open-source code as a starting point, we port the algorithm to a bare-metal platform. In their algorithm, tasks from all pending applications are issued to per-slot priority queues in a round-robin fashion; the tasks are issued to the priority queue of the slot with the fewest waiting tasks. Within the priority queues, tasks are sorted by their priority level. Lastly, we evaluate our Nimblock scheduling algorithm.

All algorithms are evaluated on the same set of stimuli. We carry out sequences of randomly selected events, where each sequence consists of 20 randomly selected events from the application pool. Each event is generated with an arrival time, batch size, and priority level. These event attributes are also randomly generated. The maximum batch size for an event is 30. The priority levels are 1, 3, 9, corresponding to low, medium, and high priorities, as discussed in Section 4.1.

To carry out a fair comparison, we run each algorithm through the same test of 10 distinct event sequences. In order to demonstrate response time reductions on a per-application basis, we compare an event's response time for each algorithm against its baseline response time and calculate the relative reduction. This allows us to get a normalized distribution, as it takes into account the disparity in application runtime. Combining the response time reductions from all individual events in the testing stimuli produces a dataset that we can draw performance conclusions from.

Response time performance numbers are measured from the moment an application enters the pending application queue to the moment the application exits the candidate application queue using the CPU clock. Because these numbers are from the hypervisor's perspective, they may include additional overhead from scheduler actions if the hypervisor is busy when an application completes. The scheduling interval at which slot reallocation is triggered is 400ms. On our hardware system, partial reconfiguration of a slot

takes, on average, around 80*ms*. The runtime of a task on a single slot varies greatly depending on the application, arrival times of other applications, and preemption. With the baseline "no-sharing" algorithm, we see that the runtime of some tasks is as small as 20% of the partial reconfiguration time. Longer tasks may take up to 200× the time of partial reconfiguration. However, many task runtimes fall between this range, and consequently, we see that masking the latency of partial reconfiguration is crucial to performance. In order to evaluate our algorithm under different congestion conditions, we run three sets of tests. In the first set of tests, we generate events that arrive with moderate delay between them. The delay is between 1500*ms* and 2000*ms*; this case emulates low-demand behavior where tasks have great opportunity to leverage additional resources. In the second set of tests, we assess our algorithm under stressful conditions, evaluating a rapid-stream of events with little delay between them. In this scenario, the delay is between 150*ms* and 200*ms*. In the last set of tests, we test a sequence where we see a consistent delay of 50*ms* between events. Here, we intend to emulate a scenario where input data is available to the FPGA in a real-time or streaming manner with a consistently short amount of time in between inputs.

5.2 Response Time Reduction

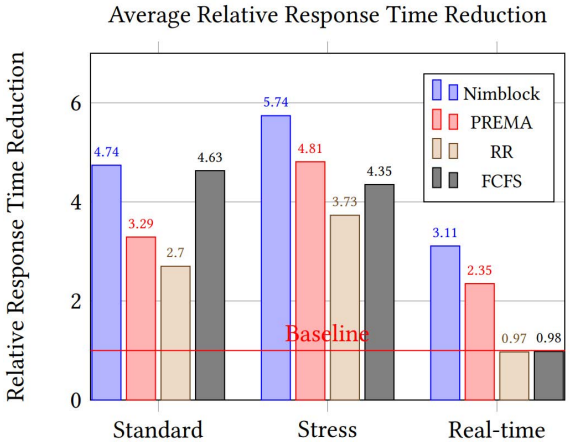


Figure 5: Relative response time reduction under different congestion conditions, normalized to the baseline.

We analyze the data using the average of the response times of the evaluated events under the three scenarios described above: standard, stress, and real-time. As seen in Figure 5, Nimblock demonstrates a 4.7× reduction in response time in the standard test, outperforming all other evaluated approaches as seen by purple bars. Nimblock's execution time is 1.4× better than PREMA, on average. Next, we consider the stress test. On average, Nimblock demonstrates a 5.7× reduction in response time over the baseline while PREMA, RR, and FCFS only reduce average response time by 4.8×, 3.7×, and 4.3×, respectively. The real-time test shows that Nimblock and PREMA vastly outperform RR and FCFS. Meanwhile, Nimblock outperforms PREMA with a 3.1× better performance over the baseline, while PREMA is only 2.4× better. RR and FCFS perform slightly worse than the baseline in this scenario. FCFS and RR are unable to fairly balance allocations for applications, resulting in poor performance in the stress and real-time tests. PREMA performs better, but since it prioritizes shorter running-applications, its performance suffers on average.

5.3 Tail Response Time

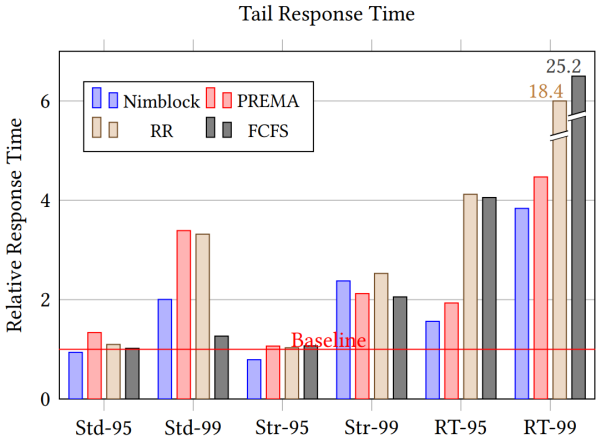


Figure 6: Tail response time under different congestion conditions, normalized to the baseline.

Tail response time is another important metric to explore when evaluating the different algorithms. We capture the tail response times by looking at the 95th and 99th percentile response time reduction numbers for all scheduling algorithms as seen in Figure 6. In the standard test (Std-95, Std-99), Nimblock demonstrates 95th percentile tail response times 1.4× lower than PREMA and 1.2× lower than RR. While at the 99th percentile, Nimblock demonstrates tail response times about 1.7× lower than PREMA and RR. In the stress test (Str-95, Str-99), Nimblock provides 95th percentile response times 1.3× lower than PREMA and RR. At the 99th percentile, Nimblock and RR demonstrate response times about 1.1× worse than FCFS and PREMA. In the real-time test (RT-95, RT-99), Nimblock has 95th percentile tail response times 1.24× better than PREMA and 2.6× better than RR and FCFS. At the 99th percentile, Nimblock sees 4.8× and 6.6× better tail response times than RR and FCFS and outperforms PREMA with 1.2× lower tail response times. Overall, Nimblock provides the lowest 99th percentile response time under real-time conditions and is better or comparable to PREMA and RR for standard and stressful conditions. Nimblock performs better than all other algorithms at the 95th percentile.

We observe that FCFS performs better than or comparably to the other scheduling algorithms for the standard and stress tests. Because FCFS is not deadline aware and does not consider priorities, it can perform well on tail response time by scheduling the oldest application. However, FCFS performs drastically worse in the real-time case. It is also important to note that FCFS does not see a competitive average performance improvement, as seen in Figure 5, and it is incapable of correctly managing deadlines as we discuss in the following section. We also see PREMA performing similarly to FCFS in the stress test. This is because PREMA chooses shorter

applications to run first when available, and response time increases are more apparent in shorter running applications. This method improves tail response times for the stress test but does not provide a significant advantage in other scenarios.

5.4 Deadline Analysis

In addition to exploring average and tail response times, we explore the deadline violation rate of the scheduling algorithms. To perform deadline analysis, we first generate an application's *single-slot latency*, the latency of the application when given a single slot to execute on with no resource contention or waiting times. We then define an application's deadline as the deadline scaling factor, D_s, multiplied by the application's single-slot latency. Because there are a variety of deadlines that could be set for a given application, we sweep D_s values from 1 to 20 at 0.25 intervals. This approach is consistent with the service-level agreement analysis performed by Choi and Rhu [3]. An application fails to meet its deadline if its response time is greater than the deadline time. For the purposes of this study, we consider high-priority applications to have tight deadlines and focus our analysis there. We assess each algorithm's 10% error point, marked on its corresponding line with a round dot, as well as its deadline violation rate at tight deadlines.

Nimblock's batch-preemption mechanism has a significant impact in reducing the amount of deadline violations for high-priority applications. When executing the standard test, we can achieve a 49% reduction in failure rate when compared to PREMA and RR as seen in Figure 7a under the tightest constraint. The tightest constraint, where $D_s = 1$, is marked with a dashed gray line. All other algorithms have a 100% violation rate at the tightest deadline. Note that as D_s increases, meeting deadlines becomes easier and we expect any effective algorithm to reach 0% failure rate as a saturation point.

When we consider the stress test, Nimblock produces fewer deadline violations than competing algorithms as shown in Figure 7b. At tight deadlines, Nimblock provides 44% fewer deadline violations than all other algorithms—consistent with its performance in the standard test. RR and FCFS perform significantly worse under the stress test. As D_s increases, Nimblock continues to outperform PREMA, reaching the 10% error point at $D_s = 3.5$ compared to $D_s = 6.0$ for PREMA.

The real-time test displayed in Figure 7c shows that while all five algorithms perform poorly at the tightest deadline, Nimblock still has a 14.3% lower violation rate than the other algorithms. As D_s increases, Nimblock instantly outperforms PREMA and RR. Nimblock's largest violation rate reduction over PREMA is 32% at $D_s = 1.75$, marked by the dashed yellow line. Nimblock provides 46% fewer errors than RR at $D_s = 3.5$, marked by the dashed pink line. Nimblock's 10% error point, $D_s = 4.25$, is also lower than PREMA's, $D_s = 5.75$.

In all three scenarios, Nimblock offers the lowest deadline violation rate at tight deadlines, and reaches the 10% error point earlier than competing algorithms in the stress and real-time tests. FCFS and RR are not priority aware, so in scenarios where more deadline violations are seen, their error rates are high. PREMA sees fewer deadline violations, but its performance falls short of Nimblock's because of its inability to preempt long-running or over-consuming applications in favor of higher priority applications. Additionally, long running tasks do not see an improvement with PREMA.

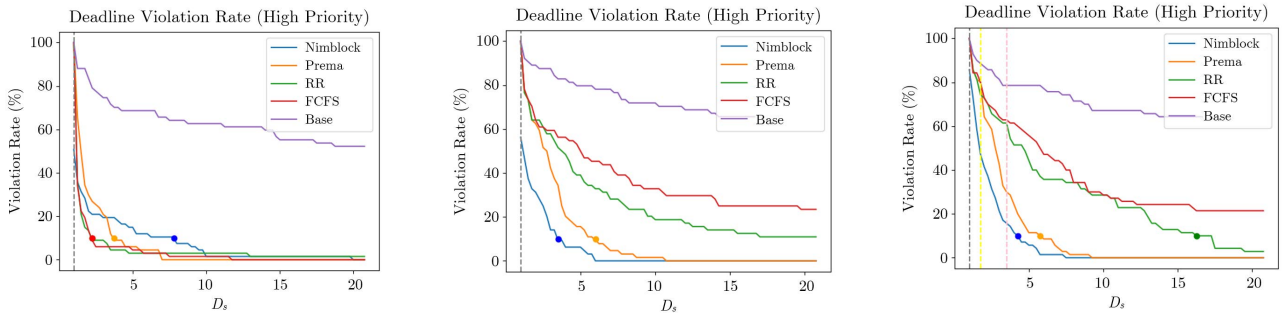
Table 3: Benchmark Latencies and Response Times

	Execution Time (s)	Response Time (s)		
Benchmark	Baseline			
LeNet	0.73	354.39		
AlexNet	65.44	274.13		
Image Compression	0.56	214.83		
Optical Flow	22.91	331.88		
3D Rendering	1.55	246.94		
Digit Recognition	984.23	984.85		
	Response Time (s)			
Benchmark	Nimblock	PREMA	RR	FCFS
LeNet	1.73	1.84	22.41	1.30
AlexNet	63.48	71.01	71.62	66.77
Image Compression	1.41	2.54	1.27	0.94
Optical Flow	14.35	31.32	29.32	31.03
3D Rendering	2.94	3.17	20.31	2.27
Digit Recognition	986.86	988.12	987.04	986.41

5.5 Benchmark Characteristics

To demonstrate the variation in the characteristics of the benchmarks we have chosen, we present the absolute response times of the six benchmarks under a test sequence with a fixed batch size of 5 where events have 500ms of delay between them. We present the execution and response times for each benchmark under the baseline no-sharing algorithm on the top half of Table 3. Response time numbers include the reconfiguration and execution time of the application. Execution time refers to the time the application spends running on the FPGA, which starts when at least one task starts running on the FPGA and ends when the last task completes. Since some tasks overlap, this is not the sum of the run times for each task. The execution times provide a reference for how long each benchmark runs relative to the others. The response times show the performance of the applications under a no-sharing algorithm and demonstrate the room for improvement. We also show the response times of each benchmark under the sharing algorithms. We see that Nimblock and FCFS perform similarly for short-running applications, outperforming PREMA and RR. For longer-running applications, we see that Nimblock generally performs better that the other algorithms.

Figure 8 shows a breakdown of the total time each application takes under the Nimblock scheduling algorithm. Run time includes the running time of all tasks summed together for one application. The total partial reconfiguration time is shown for comparison. Note that the Run time and PR time may be longer than the execution time, as the execution time includes application tasks running simultaneously. Lastly, the Wait time refers to the amount of time an application spends waiting in the queue before it starts running on the FPGA.



(a) Standard Test **(b) Stress Test** **(c) Real-time Test**

Figure 7: Deadline failure rate.

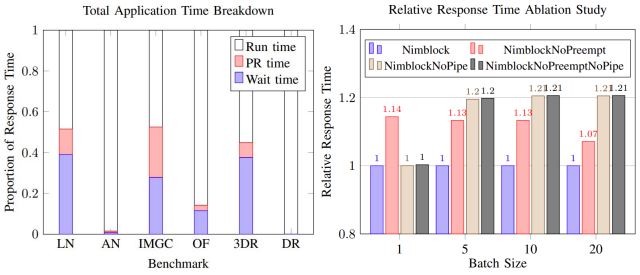


Figure 8: Run time, partial reconfiguration time, and wait time as a proportion of the total application time.

Figure 9: Relative response time for stress test under different batch sizes, normalized to the original Nimblock algorithm.

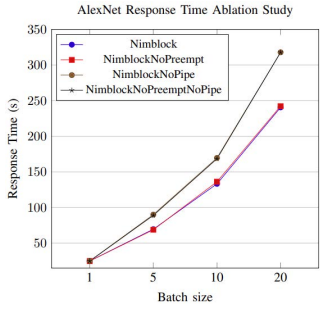


Figure 10: AlexNet response time under different batch sizes.

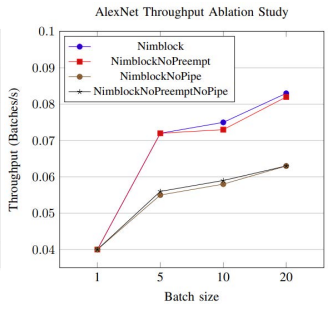


Figure 11: AlexNet throughput under different batch sizes.

5.6 Ablation Study

We analyze the impact of preemption and pipelining on the performance of the system through an ablation study. We run this test under the conditions of the stress test conducted in Section 5.2 while using fixed batch sizes of random benchmark applications with random priorities. We remove the pipelining and preemption mechanisms individually from the Nimblock algorithm, as well as both preemption and pipelining, and repeat the test. Results are normalized to the Nimblock algorithm, as shown in Figure 9.

We see that without preemption, the algorithm consistently performs $1.07 - 1.14\times$ worse than the original algorithm. Even with a batch size of 1, removing preemption when enabling pipelining affects the response times. Specifically, we observe that short-running applications benefit from preemption because allowing applications to pipeline prompts long-running application to over-consume slots for parallelism. As we previously discussed in Section 3.2, this is one of the motivating factors for including the preemption scheme. Without pipelining, the algorithm performs $1.2\times$ worse. Without preemption or pipelining, the performance is only marginally worse. This is because without making use of pipelining, other applications do not tend to monopolize resources anyway. Therefore, long-running applications do not see much impact when removing preemption after removing pipelining as well. The results show that both preemption and pipelining play a part in Nimblock's performance improvements over the other sharing algorithms.

We use AlexNet to further inspect the effects of preemption and pipelining. As shown in Table 3, AlexNet takes a moderate amount of execution time and sees varying results over different scheduling algorithms. We consider its response times in the above experiment, shown in Figure 10. As in Figure 9, we find that removing pipelining impacts the performance most negatively. Without pipelining, preemption does not provide as large an improvement, shown by the overlapping results for NimblockNoPipe and NimblockNoPreemptNoPipe. Note that for a batch size of 1, removing preemption or pipelining does not impact the performance. For larger batch sizes, we see that removing the preemption scheme does not drastically degrade the performance. Since AlexNet is a longer-running application that can monopolize resources, we would instead expect that shorter-running applications such as LeNet, Image Compression, and 3D Rendering would see a larger improvement from preemption. Due to parallelization across multiple slots, we see that the response time is not linear with regard to batch size. This is apparent in Figure 11 where we measure the throughput of AlexNet under different batch sizes. We see in Figure 11 that Nimblock and NimblockNoPreempt, which enable pipelining, demonstrate a higher throughput for AlexNet. As we increase

the number of batches past 5, we don't see as large a jump in performance, suggesting that even at small batch sizes Nimblock makes good use of the available resources.

6 RELATED WORK

6.1 FPGA Virtualization

Numerous works have pursued FPGA virtualization in recent years. The authors of AMORPHOS [19] proposed a novel system which combines bitstreams at the server-side to enable sharing of FPGAs and reduce response times. Their approach places bitstream generation on the critical path and only explores coarse-grained FPGA sharing. Other works such as ViTAL [35] and Hetero-ViTAL [36] allow for fine-grained resource sharing in a comprehensive flow, but do not explore advanced scheduling opportunities through preemption and pipelining. Hetero-ViTAL extends ViTAL by scaling out to heterogeneous classes of devices through a two-level ISA.

Additional works have explored adding operating system-like capabilities to FPGAs [21, 24, 37], adding support for shared memory access, networking, or peripheral access. Of these approaches, Coyote [21] also employs tiling and enables full networking and virtual memory stacks. Coyote explores simple round-robin scheduling schemes but does not seek to extract additional performance from the virtualized system. The Optimus system [24] employs tiling and explores scheduling optimizations including preemption. However, it does not support partial reconfiguration and its version of preemption time-multiplexes a single shared application, which occupies the entire FPGA, among multiple users. Our work instead focuses on using dynamic partial reconfiguration, which allows for arbitrary applications from different users sharing the same FPGA. Therefore, multiple tasks may continue running while a task running on a single slot is able to be preempted.

6.2 Scheduling

Leveraging DPR to improve performance and task scheduling on FPGAs has been explored in many paradigms to find optimal, often ILP-based solutions to the scheduling problem [6, 11, 27]. In general, these approaches explore specific optimizations or improvements of the ILP formulation or solver. DML [12] builds on this to show that pipelining and fine-grained sharing can hide reconfiguration time on FPGA devices and works on a variety of real-world workloads by exploring optimizations around pipelining and batching. DML evaluates the effect of increasing the batch size and different batching strategies on the effect of reconfiguration. However, DML uses an expensive ILP solver on the critical path to find an optimal scheduling solution, making it difficult to scale to large numbers of applications with larger batch sizes. Moreover, DML relies on prior knowledge of applications and their arrival times, and it disregards application priority levels. Both DML and Nimblock schedulers use the concept of independent-scheduling for allotting slots, but while Nimblock performs the allocation dynamically and without any need for user input, DML requires the user to statically designate a certain number of slots to each application. Because DML does not implement a hypervisor to run applications as they arrive at a system, it is ill-suited to real-time scheduling.

Similarly, a great deal of work has been done in the real-time scheduling space, particularly when considering priorities. PREMA [3] considers time-multiplexing of Neural Processing Unit (NPU) accelerators in the cloud to reduce application response time in the face of varying priority levels. The authors evaluate the performance of PREMA with different batch sizes. Unlike our approach, they do not consider reconfigurable hardware or fine-grained space-sharing within their scheduling algorithm. Their time-multiplexing preemption mechanism performs in a different context than ours, as they explore preemption on a single NPU, rather than a device with multiple slots for processing. Another study [34] uses PREMA scheduling techniques to reduce response time for FPGA-based DNN accelerators in a cloud setting. Their approach uses PREMA to select tasks to run and then solves an optimization problem to assign slots on FPGAs. Unlike our approach, their approach relies on accelerating only DNN layers with known and consistent computation patterns, and does not consider pipelining or preemption as a resource sharing technique. Our approach allows for arbitrary logic in reconfigurable regions and explores batch-preemption to reduce deadline violations.

7 CONCLUSION AND FUTURE WORK

As FPGA virtualization techniques become more common across the cloud and edge, it is important to leverage the fine-grained sharing capabilities to improve application response times and increase the efficiency of hardware utilization. We demonstrate up to a 5.7× performance improvement in average application response time when compared to non-sharing techniques and up to a 2.1× average improvement in performance over competitive scheduling algorithms such as PREMA when evaluated on actual hardware with a varied, real-world benchmark suite. Moreover, we make a case for enabling preemption on reconfigurable hardware to enable aggressive optimizations with the ability to roll back. Preemption enables improvements on existing scheduling solutions, and we demonstrate up to a 2.6× reduced tail response time and up to a 49% reduction in deadline violations when compared to other real-time scheduling algorithms.

Hypervisors such as Nimblock are essential to enable performant virtualization schemes, and low overhead solutions must exist without solving expensive ILP problems. At its core, our approach is device agnostic, though certain architectural features in the FPGA could enable further optimizations and make our solution more effective. For example, a NoC would allow for optimized data transfer between slots; the current design requires slots to communicate through the ARM core. Additionally, architectural modifications which would enable preemption at a finer granularity, such as increased on-chip memory and state registers, could also improve performance. Exploration of larger classes of devices such as cloud-scale FPGAs, smaller edge-scale FPGAs, or ACAP platforms will help pave the way for additional virtualization exploration. We have open-sourced Nimblock to motivate further works[1].

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their helpful feedback. This work was supported by the AMD Center of Excellence, the AMD HACC Initiative, and the IBM-Illinois Discovery Accelerator Institute.

[1] https://github.com/UIUC-ChenLab/Nimblock

A ARTIFACT APPENDIX

A.1 Abstract

Nimblock is a hypervisor for fine-grained sharing on a slot-based FPGA overlay. This artifact contains the prototype software implementation of the Nimblock hypervisor on a bare-metal embedded processor, along with the necessary hardware description files and bitstreams required to run the scheduler on the system described in the evaluation. We describe the workflow used to obtain the results in the evaluation.

A.2 Artifact check-list (meta-information)

- **Program:** Nimblock
- **Binary:** Compiled in Xilinx SDK.
- **Data set:** Applications from the Rosetta benchmark suite and custom benchmark applications (HLS code provided).
- **Run-time environment:** Xilinx SDK 2019.1
- **Hardware:** Xilinx ZCU106 FPGA is needed to replicate results with provided bitstreams and SDK libraries.
- **Execution:** Test sequences can each take 30 min to run.
- **Metrics:** Response time (latency)
- **Output:** Performance reports for test cases described in evaluation.
- **Experiments:** All software and bitstreams required to replicate test cases in the evaluation are included. Scripts and instructions to generate custom test sequences are also included.
- **How much disk space required (approximately)?:** 15GB (if Vivado/SDK not yet installed). 150 MB for SDK workspace.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 12 hours
- **Publicly available?:** Yes.
- **Code licenses:** Apache-2.0 License
- **Archived DOI:** 10.5281/zenodo.7818841

A.3 Description

A.3.1 How to access. The source code and bitstreams are available on GitHub: https://github.com/UIUC-ChenLab/Nimblock and are archived on Zenodo: https://zenodo.org/record/7818841. The repo contains a .zip file which can be directly imported into Xilinx SDK, as well as the original source code for the Nimblock software. Scripts to generate test sequences can also be found in the GitHub repository.

A.3.2 Hardware dependencies. A Xilinx ZCU106 FPGA is required to use the provided source code and bitstreams. It must be programmed via JTAG. Other FPGAs are not currently supported.

A.3.3 Software dependencies. Xilinx SDK 2019.1 is required to run the software on the embedded ARM core. Other versions of SDK have not been tested with the provided code and bitstreams.

A.4 Installation

Xilinx SDK 2019.1 must be installed on the host machine. Setup information is provided in the repository README. Bitstreams are already generated, so Vivado installation is not needed. The FPGA must be programmed over JTAG, and the software project to be imported into the SDK workspace is provided.

A.5 Experiment workflow

All necessary experiments can be run through Xilinx SDK. New test cases can be generated with provided python scripts. A .zip file is provided to import into SDK. The SD card on the ZCU106 must be loaded with partial bitstreams, which are also provided. The FPGA can be programmed through SDK. The hypervisor software is run bare-metal on the embedded core through SDK. Reports can be collected over a serial console. More details are provided in the repository.

A.6 Evaluation and expected results

We provide scripts to generate randomized test sequences for each test case described in our evaluation. The scripts can be modified to generate shorter test sequences. The sequences can be copied into the source code for the testbed in SDK. Reports with average, median, and tail response times can be generated, as well as deadline violation rates. More detailed instructions are provided in the repository. Serial output can be saved and parsed with provided scripts to generate reports for response times and deadline violation rates.

A.7 Experiment customization

The delay between events can be adjusted with the event generation script, as can the application priorities and batch sizes. Test sequences can also be manually created with the desired applications. In addition to being able to customize test sequences, the scheduling algorithm(s) can easily be modified in software.

REFERENCES

[1] Amazon. 2022. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[2] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*. 126–131. https://doi.org/10.1109/FPL.2009.5272532

[3] Y. Choi and M. Rhu. 2020. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 220–233. https://doi.org/10.1109/HPCA47549.2020.00027

[4] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[5] Simone Corbetta, Massimo Morandi, Marco Novati, Marco Domenico Santambrogio, Donatella Sciuto, and Paola Spoletini. 2009. Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 11 (2009), 1650–1654. https://doi.org/10.1109/TVLSI.2008.2005670

[6] Roberto Cordone, Francesco Redaelli, Massimo Antonio Redaelli, Marco Domenico Santambrogio, and Donatella Sciuto. 2009. Partitioning and Scheduling of Task Graphs on Partially Dynamically Reconfigurable FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 5 (2009), 662–675. https://doi.org/10.1109/TCAD.2009.2015739

[7] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. Association for Computing Machinery, New York, NY, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[8] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 217–226. https://doi.org/10.1145/3020078.3021739

[9] Andrea Damiani, Giorgia Fiscaletti, Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. 2022. BlastFunction: A Full-Stack Framework Bringing FPGA Hardware Acceleration to Cloud-Native Applications. *ACM Trans. Reconfigurable Technol. Syst.* 15, 2, Article 17 (jan 2022), 27 pages. https://doi.org/10.1145/3472958

[10] Khoa Dang Pham, Edson Horta, and Dirk Koch. 2017. BITMAN: A tool and API for FPGA bitstream manipulations. In *Design, Automation Test in Europe Conference Exhibition, 2017*. 894–897. https://doi.org/10.23919/DATE.2017.7927114

[11] Enrico A. Deiana, Marco Rabozzi, Riccardo Cattaneo, and Marco D. Santambrogio. 2015. A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. https://doi.org/10.1109/ReConFig.2015.7393328

[12] Ashutosh Dhar, Edward Richter, Mang Yu, Wei Zuo, Xiaohao Wang, Nam Sung Kim, and Deming Chen. 2022. DML: Dynamic Partial Reconfiguration With Scalable Task Scheduling for Multi-Applications on FPGAs. *IEEE Trans. Comput.* 71, 10 (2022), 2577–2591. https://doi.org/10.1109/TC.2021.3137785

[13] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 797–813. https://doi.org/10.1145/3503222.3507732

[14] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3431920.3439289

[15] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3490422.3502361

[16] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[17] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2749469.2750412

[18] Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, and Ce Zhang. 2017. FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-Off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 160–167. https://doi.org/10.1109/FCCM.2017.39

[19] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 107–127. http://www.usenix.org/conference/osdi18/presentation/khawaja

[20] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. 2018. GPU Enabled Serverless Computing Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 533–540. https://doi.org/10.1109/PDP2018.2018.00090

[21] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[22] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 203–215. https://doi.org/10.1109/TCAD.2006.884574

[23] André Lalevée, Pierre-Henri Horrein, Matthieu Arzel, Michael Hübner, and Sandrine Vaton. 2016. AutoReloc: Automated Design Flow for Bitstream Relocation on Xilinx FPGAs. In *2016 Euromicro Conference on Digital System Design (DSD)*. 14–21. https://doi.org/10.1109/DSD.2016.92

[24] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[25] Microsoft. 2018. Real-time AI: Microsoft announces preview of Project Brainwave. https://blogs.microsoft.com/ai/build-2018-project-brainwave/

[26] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated Serverless Computing Based on GPU Virtualization. *J. Parallel Distrib. Comput.* 139, C (may 2020), 32–42. https://doi.org/10.1016/j.jpdc.2020.01.004

[27] Andrea Purgato, Davide Tantillo, Marco Rabozzi, Donatella Sciuto, and Marco D. Santambrogio. 2016. Resource-Efficient Scheduling for Partially-Reconfigurable FPGA-Based Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 189–197. https://doi.org/10.1109/IPDPSW.2016.176

[28] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22. https://doi.org/10.1109/MM.2015.42

[29] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. 2020. A Survey of System Architectures and Techniques for FPGA Virtualization. https://doi.org/10.48550/ARXIV.2011.09073

[30] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A Case for Function-as-a-Service with Disaggregated FPGAs. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 333–344. https://doi.org/10.1109/CLOUD53861.2021.00047

[31] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-Sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. Association for Computing Machinery, New York, NY, USA, 16–25. https://doi.org/10.1145/2847263.2847276

[32] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. 2019. A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities. https://doi.org/10.48550/ARXIV.1901.04988

[33] Xilinx 2019. *Vivado Design Suite User Guide: Partial Reconfiguration.* Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug909-vivado-partial-reconfiguration.pdf

[34] Shulin Zeng, Guohao Dai, Hanbo Sun, Jun Liu, Shiyao Li, Guangjun Ge, Kai Zhong, Kaiyuan Guo, Yu Wang, and Huazhong Yang. 2022. A Unified FPGA Virtualization Framework for General-Purpose Deep Neural Networks in the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 24 (dec 2022), 31 pages. https://doi.org/10.1145/3480170

[35] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 845–858. https://doi.org/10.1145/3373376.3378491

[36] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 470–483. https://doi.org/10.1109/ISCA52012.2021.00044

[37] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 22, 7 pages. https://doi.org/10.1145/3124680.3124743

[38] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 269–278. https://doi.org/10.1145/3174243.3174255