NIMBLOCK: SCHEDULING FOR FINE-GRAINED FPGA SHARING THROUGH VIRTUALIZATION

BY

PAUL RECKAMP

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Adviser:

Professor Deming Chen

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-multiplex 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. 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 9× lower median response times when compared to the baseline scheduling algorithms. We additionally demonstrate up to 21% fewer deadline violations and up to 2.1× lower tail response times when compared to other high-performance algorithms.

To my parents, siblings, and friends for their love and support.

ACKNOWLEDGMENTS

Thank you to Paul Hartke, Samuel Bayliss, and Zachary Blair of Xilinx for assisting with technical issues throughout this project. Thank you to all of the members of my research group for providing support and guidance during the unprecedented times of a global pandemic. Finally, thank you to my adviser, Professor Deming Chen, who pushed me to achieve more, and whose guidance was instrumental in my success.

TABLE OF CONTENTS

CHAPTER 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 [1, 2, 3, 4, 5, 6] more flexibly and at lower power than CPUs and GPUs [7, 8, 9]. 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 [10] or Microsoft Azure [11].

CPU compute resources have traditionally been virtualized through 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 timeslots on cores to an application. Unlike CPUs, FPGAs are generally programmed with a single bitstream requiring independent users to cooperate to program the device [12] or wait their turn in line. 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. Because of these multiplexing issues, increased demand for FPGA compute resources in the cloud is satisfied by increasing the number of FPGAs and implementing queues—AWS EC2 F1 instances [10], for example, provide FPGAs in totality to the end user.

FPGA virtualization strives to bring virtualized features to FPGAs 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.

Canonical solutions for FPGA virtualization [13, 14] create a custom FPGA overlay that splits the fabric into *slots*, independently reconfigurable blocks (or tiles) that are able to host arbitrary logic in the system. 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 and in a fraction of the time of regular reconfiguration [15].

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 [16, 17, 18, 19]. 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 datacenter or real-time system, applications often have different priorities or deadlines to meet. Finally, the scheduler should consider performance optimizations within an application such as pipelining across batches and hiding reconfiguration time behind active computation. 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 considerations for arbitrary real-time workloads and priority levels. Nimblock employs an overlay architecture with reconfig-

urable 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. We summarize our contributions as:

1. We present a novel scheduling algorithm that considers applications with arbitrary arrival times and priorities and enables pipelining and preemption. We evaluate our algorithm on a ZCU106 FPGA and validate its efficacy on real workloads.

2. We demonstrate up to a 9× median response time improvement over a baseline scheduling approach and up to a 2.1× lower tail response times compared to other high-performance algorithms.

3. We achieve up to a 21% lower deadline violation rate than previous algorithms.

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 Nimblock System

2.1.1 Overlay

The Nimblock overlay, inspired by the overlay used in [20], 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 2.1.

At the core of the PS portion of the overlay is the embedded ARM core, which runs the Nimblock hypervisor and 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 then access the partial bitstream and send a request to the aonfiguration 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.

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

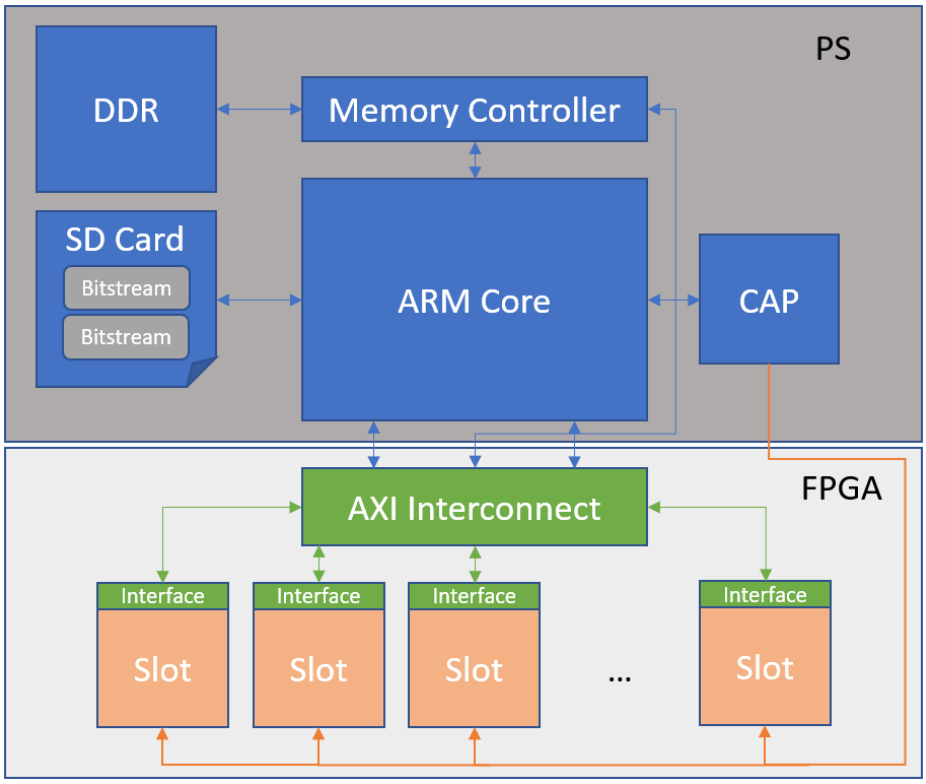

Figure 2.1: Nimblock overlay. Static elements are in green, reconfigurable elements are in orange.

as decoupling resources to isolate configurable logic during reconfiguration. The reconfigurable slots are floorplanned to have the same resource size and can host arbitrary user logic. For our purposes, the logic is configured to have a single memory-mapped interface for control and second memory-mapped interface for data—easily achievable through high-level synthesis flows using interface pragmas.

2.1.2 Hypervisor

As system manager, the Nimblock hypervisor runs on the embedded ARM core, driving reconfiguration, managing application data, and running the scheduling algorithm outlined in Section 2.3. 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 with nodes representing tasks and edges representing dependencies. For example, if the application is Lenet, it could be split into three tasks, each a grouping of two layers, and two edges linking the three nodes in a chain. The Nimblock compilation flow is agnostic to partitioning method, and would function with automatic task partitioning and compilation methodologies [14, 21, 22] 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 for each slot. Partial bitstream relocation can reduce the number of bitstreams we need to store and generate, but exploration of this concept [23, 24, 25] is beyond the scope of this work. The bitstreams are added to the hypervisor with a header to provide 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 can easily parse the information from a JSON file. For the purpose of this work, we define batch 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 [20].

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. 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.

2.2 Motivation

2.2.1 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 application's retirement time, R_i. 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.

2.2.2 Sharing Modes

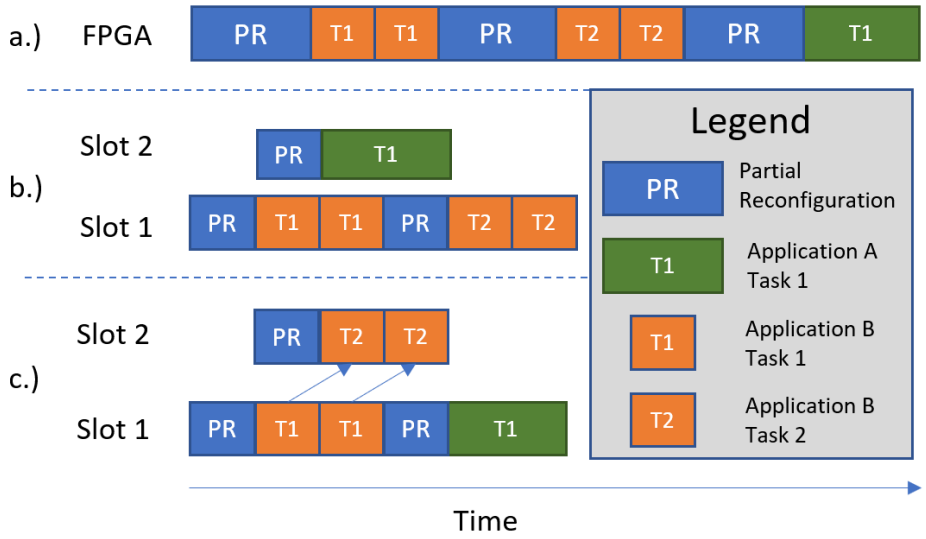

Figure 2.2: Sharing modes for multiple FPGA slots. Arrows represent dependencies across pipeline stages.

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 2.2(a). By partitioning the FPGA into slots, task-level parallelism is extracted from independent tasks. This approach overlaps reconfiguration time with computation and enables multiple applications' tasks to execute simultaneously as seen in Figure 2.2(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. In this mode, an application's tasks can co-exist on the FPGA while working on different batch items. 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 2.2(c).

Pipelining allows us to extract additional performance and reduce the response time of applications, but requires a single application to consume multiple slots simultaneously. Rampart pipelining can choke performance for later arriving applications, 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. Later 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.

The decision to add preemption 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 a performance decrease if implemented poorly and is unpopular among current approaches [18, 17].

Moreover, checkpointing arbitrary FPGA state is required for preemption, but checkpointing is challenging—user state includes stateful logic blocks in the FPGA which are difficult to capture [18]. Despite these challenges, the Nimblock scheduling algorithm employs fine-grained sharing, pipelining, and preemption to improve application performance.

2.3 Scheduling Algorithm

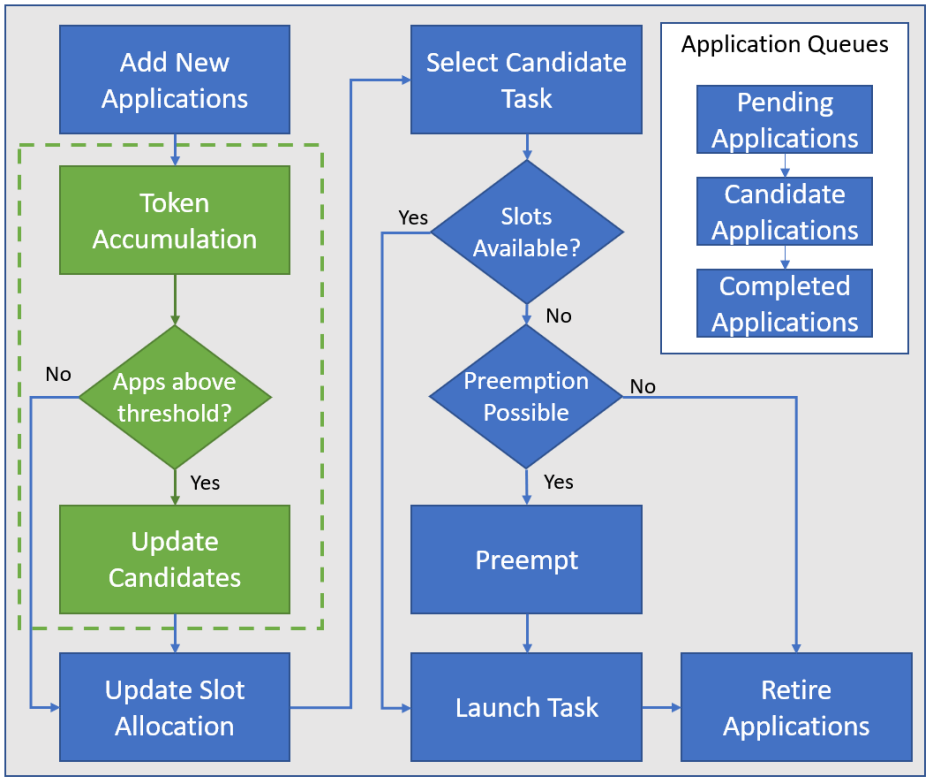

Figure 2.3: Block diagram of the Nimblock scheduling algorithm. Green denotes elements borrowed from PREMA [16].

Figure 2.3 presents a block diagram showing the major operational steps of our scheduling algorithm. First, all arriving applications are added to the scheduler and initialized. Second, applications accumulate tokens and the candidate application pool is updated (Section 2.3.1). Third, we update the slot allocation for each candidate application (Section 2.3.2). We then select a task to schedule (Section 2.3.3) and a slot to reconfigure (Section 2.3.4). Lastly, the selected task is launched and we retire any completed

applications.

2.3.1 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 execute quickly. To balance these factors, we utilize the token accumulation strategy from the PREMA scheduling algorithm [16, 17]. Consistent with previous work, our implementation uses three priority levels of increasing 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 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 serialize the application's task-graph and sum task latency estimates to obtain an application latency estimate.

Algorithm 1 Candidate selection

initialization
for application $a \in$ arrival queue A **do**
  $a.token \leftarrow a.priority$
end for
for application $a \in$ pending queue R **do**
  $a.token \leftarrow a.token + \alpha \times a.priority \times degradation_{norm}$
end for
$threshold \leftarrow \max([floor_{prio}(a.token) \text{ for } a \in R])$
$candidates \leftarrow [a \text{ for } a \in R \text{ if } a.token > threshold]$
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. As applications accumulate tokens, they increase the candidate threshold using the PREMA thresholding method. 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.

2.3.2 Slot Allocation

Whenever the candidate application pool changes and at periodic scheduling intervals, the Nimblock scheduling algorithm triggers a *reallocation* where it decides how to allocate slots in a way that reduces response times and ensures deadlines are met. Our slot allocation algorithm, shown in Algorithm 2, 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.

Algorithm 2 Slot Allocation

```
candidates ← sort_age(candidates)
for candidate c ∈ candidates do
  c.slots ← 1
end for
for candidate c ∈ candidates do
  if slots available then
    c.slots ← min(c.goal, c.incomplete_tasks)
  end if
end for
if slots available then
  for candidate c ∈ candidates do
    c.slots ← c.incomplete_tasks
  end for
end if
```

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*, derived from the concept of independent-scheduling identified in DML [20]. 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.

While they used this information to reduce ILP scheduling time, they also used it to allocate slots for applications. 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 ILP formulation from DML [20] 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 [26]. 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 may happen in parallel with synthesis, place and route, and bitstream generation, keeping it firmly off the critical path of the user flow.

We then inspect the saturation points and identify common trends to set static goal numbers for each application. 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. For the purposes of this work, we found that a static, unified goal number of 3 effectively balanced individual application performance and device-sharing. We validated this by sweeping across potential goal numbers and identifying the best performing solution.

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.

2.3.3 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

Algorithm 3 Task Selection

```
candidates ← sort_age(candidates)
for candidate c ∈ candidates do
  if c.occupancy < c.allocation then
    for task t ∈ c.incomplete_tasks do
      if t.ready or t.can_pipe then
        return t
      end if
    end for
  end if
end for
```

schedule at each timestep as shown in Algorithm 3. 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.

2.3.4 Preemption

Algorithm 4 Preemption

```
overconsumption ← 0
for slot s ∈ slots do
  a ← s.application
  consumption ← a.slots_used − a.slots_allocated
  if s.waiting and consumption > overconsumption then
    overconsumption ← consumption
    overconsumer ← a
  end if
end for
tasks ← topological_sort(overconsumer.running_tasks)
preempt_task ← tasks.end()
if preempt_task.slot.waiting_for_batch then
  return preempt_task.slot
end if
```

As we saw in Section 2.3.2, slot allocations are modified as the candidate application pool changes. As additional applications arrive, we note that

some applications will be reassigned fewer slots than they are currently using. Without a method to intervene, the overconsumers will continue to utilize their slots until completion—in the worst case, this could delay the execution of other applications and increase the number of violated deadlines. To avoid resource monopolization, Nimblock introduces preemption for FPGAs, enabling both fine-grained time- and space-multiplexing in our runtime.

In the case of an application being ready to execute and there being no available slots to schedule it to, we begin our preemption algorithm as seen in Algorithm 4. Our algorithm iterates over every currently running application and evaluates its ability to be preempted. 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.

We select the candidate application that has surpassed its slot allocation by the most for preemption; 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 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 until it reaches a batch boundary.

CHAPTER 3

EVALUATION

3.1 Methodology

We evaluate our scheduling algorithm on a ZCU106 FPGA from Xilinx partitioned into ten slots, and we run our hypervisor as a baremetal 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 contains an application name, batch information, priority level, and arrival time, which it releases to the hypervisor after the event's arrival time has passed. We select applications from the suite used in [20]: 3D-rendering, digit recognition, optical flow, image compression, Lenet, and Alexnet. The former three are from the Rosetta benchmark suite [27], and the later three are custom benchmarks. We manually partitioned the benchmarks into optimized tasks that fit in a single slot on the FPGA and generated bitstreams using Vivado 2019.1. 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 2.1.2. 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 board 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 or pipeline across batches. In addition to the no-sharing baseline algorithms, we implement three task-based heuristic algorithms to compare against.

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 from in the order that they were made ready. Applications are able to pipeline and 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 [16, 17] are tuned for their target systems and do not map directly to our sharing scheme. To account for these differences, we make the following modifications to the approach in [16]. We keep the token accumulation scheme as well as the candidate selection methodology of choosing the shortest candidate to execute next. Because we target multiple slots instead of the single monolithic device used in [16], we remove the preemption scheme like the authors of [17] did. Furthermore, we allow tasks to pipeline across batches for this PREMA version. It does not include techniques that are unique to Nimblock as shown in Figure 2.3.

The third algorithm is a queue-based round-robin (RR) scheduling algorithm adapted from the implementation in [18]. Using their open-source code as a starting point, we port the algorithm to a baremetal platform and modify it to enable pipelining optimizations. 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 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, each consisting of 20 randomly selected events with randomized arrival times and batch sizes. In order to evaluate our algorithm under different congestion conditions, we run two sets of tests. In the first set of tests, we generate events that arrive with moderate delay between them; 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. To account for variation in random stimuli, we run each algorithm through many distinct event sequences. 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.

3.2 Analysis

3.2.1 Response Time Reduction

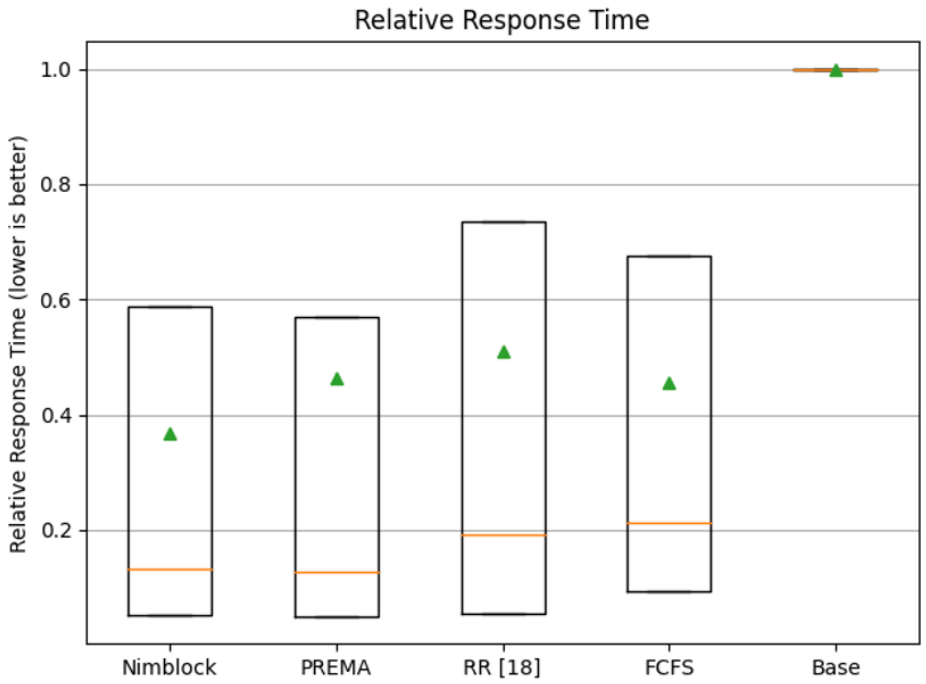

Figure 3.1: Standard test relative response time, normalized to the baseline. Median response time reduction is the orange line, with the bottom and top of the box representing the first and third quartiles respectively. The average is represented by the green triangle.

In order to demonstrate response time reductions on a per-application basis, we compare an event's response time against its baseline response time and calculate the relative reduction. Combining the response time reductions from all events from the testing stimuli produces a dataset we can draw performance conclusions from. We analyze the data using two measures of central tendency, the mean and median, and a measure of consistency or spread, the interquartile range (IQR).

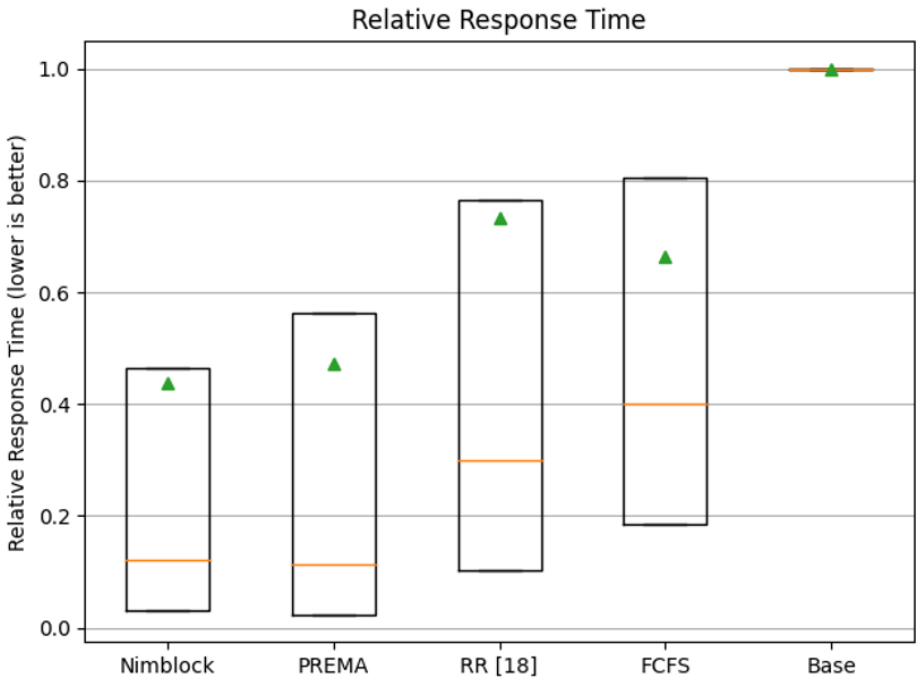

Figure 3.2: Stress test relative response time, normalized to the baseline. Median response time reduction is the orange line, with the bottom and top of the box representing the first and third quartiles respectively. The average is represented by the green triangle.

As seen in Figure 3.1, Nimblock and PREMA demonstrate nearly 8× lower response times at the median, outperforming the 5× lower from RR and FCFS under standard testing conditions compared to the baseline. On average, Nimblock provides 2.7× lower response times, outperforming all other evaluated approaches as seen by the green triangles. Nimblock, PREMA, and FCFS demonstrate the most consistent results, with IQR values between 0.53 and 0.58. Round robin, on the other hand, demonstrates a larger spread with an IQR of 0.68.

When we consider the stress test in Figure 3.2, algorithmic impact becomes increasingly clear. Under this test, the FCFS algorithm reduces median response time by only 2.5×, while the RR algorithm performs slightly better with 3.3× lower response times compared to the baseline. Both Nimblock and PREMA demonstrate performance consistent with their performance on the standard test, demonstrating 9× lower median response times. On average, Nimblock demonstrates 2.3× lower response times while PREMA, RR, and FCFS only lower average response time by 2.1×, 1.3×, and 1.5×

respectively. Moreover, the Nimblock scheduling algorithm is able to provide this reduction with more consistency than other algorithms, with an IQR of only 0.43 compared to the 0.54 of PREMA, 0.66 of RR, and 0.62 of FCFS.

3.2.2 Tail Response Time

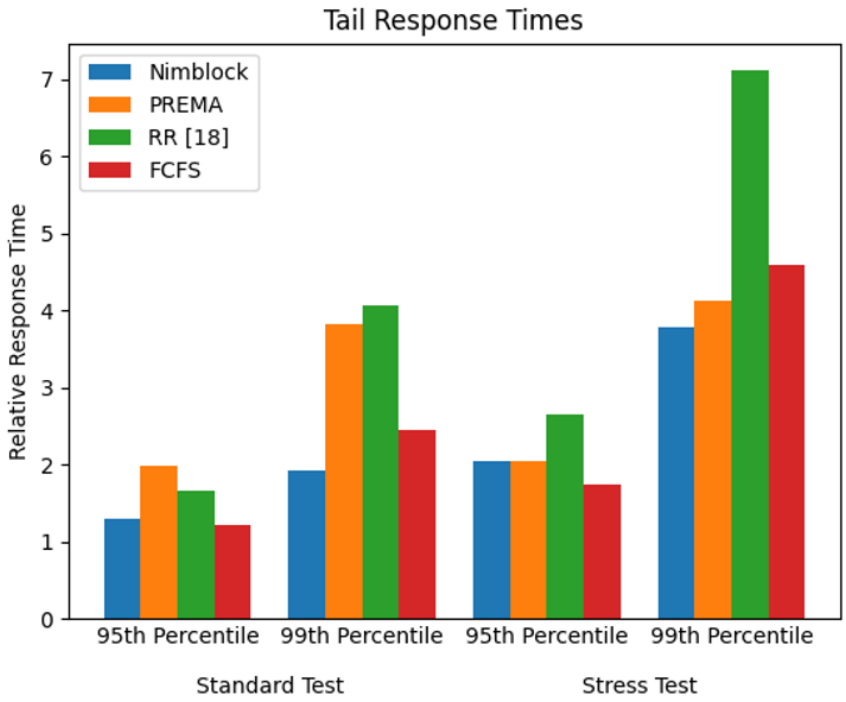

Figure 3.3: Tail response time, normalized to the baseline

Another important metric to explore response time improvements is the tail response time. We capture this by looking at the 95th and 99th percentile response time reduction numbers for all scheduling algorithms as seen in Figure 3.3. In the standard test, Nimblock demonstrates 95th percentile tail response times 1.5× lower than PREMA and 1.3× lower than RR. While at the 99th percentile, Nimblock demonstrates tail response times 2× lower than PREMA and 2.1× lower than RR. In the stress test, Nimblock provides 95th percentile response times equivalent to PREMA and 1.3× lower than RR. At the 99th percentile, Nimblock demonstrates response times 1.1× and 1.9× lower than PREMA and RR respectively. Overall, Nimblock provides the lowest 99th percentile response time and performs comparably to or better than PREMA and RR at the 95th percentile.

When comparing to FCFS, we note that Nimblock outperforms at the 99th percentile with tail response times 1.3× lower and 1.2× lower during the standard test and stress test respectively. However, at the 95th percentile, Nimblock performs comparably during the standard test and 1.17× worse during the stress test. Because FCFS is not deadline aware and does not consider priorities, it can perform well on tail response times by always scheduling the oldest application. While FCFS is better in this particular case, it is consistently worse in the average case and is incapable of correctly managing deadlines.

3.2.3 Deadline Analysis

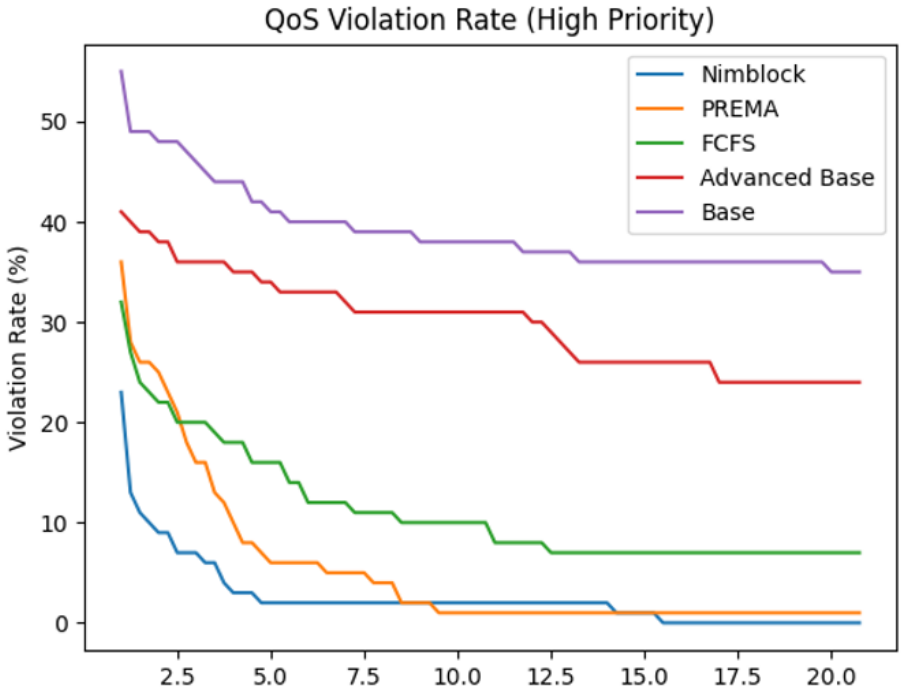

Figure 3.4: Standard test deadline failure rate.

In addition to exploring average, median, and tail response time, 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

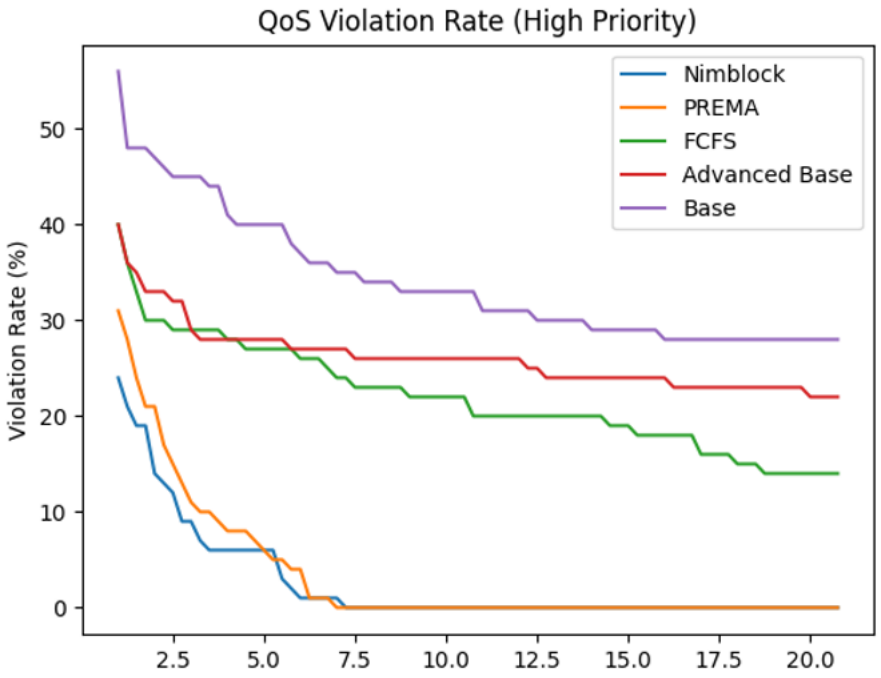

Figure 3.5: Stress test deadline failure rate.

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 [16]. 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.

Nimblock's 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 21% reduction in failure rate when compared to PREMA and a 11% reduction when compared to RR as seen in Figure 3.4. RR initially performs well at tight deadlines, but its high tail response time prevents it from achieving zero violations as D_s increases—RR reaches the 10% error point at $D_s = 18.25$ compared to $D_s = 4.5$ for Nimblock and $D_s = 6.0$ for PREMA. Note that as D_s increases, meeting deadlines becomes increasingly easy and we expect any effective algorithm to reach 0 failure rate as a saturation point.

When we consider the stress test, Nimblock continues to produce fewer deadline violations than competing algorithms as shown in Figure 3.5. At tight deadlines, Nimblock provides 19% fewer deadline violations than PREMA—

consistent with its performance in the standard test. On the other hand, RR performs significantly worse under the stress test, and Nimblock provides 30% fewer errors at tight deadlines. As D_s increases, Nimblock continues to outperform PREMA, reaching the 10% error point at $D_s = 3.75$ compared to $D_s = 4.75$ for PREMA. In both test scenarios, Nimblock offers the lowest deadline violation rate at tight deadlines, and reaches the 10% error point earlier than competing algorithms.

CHAPTER 4

RELATED WORK

4.1 FPGA Virtualization

There have been numerous works pursuing FPGA virtualization in recent years. The authors of AMORPHOS [12] proposed a novel system which combines bitstreams 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 [14] and Hetero-ViTAL [28] allow for fine-grained resource sharing in a comprehensive flow, but do not explore 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 [18, 29, 19], adding support for shared memory access, networking, or peripheral access. Of these approaches, Coyote [18] 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 [19], employs tiling and explores scheduling optimizations including preemption. However, its version of preemption time-multiplexes a single shared application among multiple users while our work enables arbitrary applications from different users to preempt.

4.2 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 [30, 31, 32]. Building on this, DML [20] demonstrated that pipelining and fine-grained sharing can hide reconfiguration time on multiple FPGA devices and works on a variety of real-world workloads. In general, these approaches explore specific optimizations or improvements of the ILP formulation or solver; only DML explores optimizations around pipelining and batching. However, DML uses an expensive ILP solver on the critical path to find an optimal scheduling solution, minimizing scaling 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. Combined, these factors make it 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[16] considers time-multiplexing of NPU accelerators in the cloud to reduce application response time in the face of varying priority levels. Unlike our approach, they do not consider reconfigurable hardware or fine-grained space-sharing within their scheduling algorithm. Another study [17] 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 preemption as a resource sharing technique. Our approach allows for arbitrary logic in reconfigurable regions and explores preemption to reduce deadline violations.

CHAPTER 5

CONCLUSIONS

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 9× performance improvement in median application response time when compared to non-sharing techniques 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 2.1× lower tail response times and up to a 21% reduction in deadline violations when compared to other real-time scheduling algorithms.

REFERENCES

[1] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[2] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, "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*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2847263.2847276 p. 16–25.

[3] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[4] G. Dai, Y. Chi, Y. Wang, and H. Yang, "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*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2847263.2847339 p. 105–110.

[5] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3020078.3021739 p. 217–226.

[6] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An appliance for big data analytics," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2750412 p. 1–13.

[7] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 126–131.

[8] T. Wang, C. Wang, X. Zhou, and H. Chen, "A survey of FPGA based deep learning accelerators: Challenges and opportunities," 2019. [Online]. Available: https://arxiv.org/abs/1901.04988

[9] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "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)*, 2017, pp. 160–167.

[10] Amazon, "Amazon EC2 F1 Instances," 2022. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

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

[12] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018. [Online]. Available: http://www.usenix.org/conference/osdi18/presentation/khawaja pp. 107–127.

[13] M. H. Quraishi, E. B. Tavakoli, and F. Ren, "A survey of system architectures and techniques for FPGA virtualization," 2020. [Online]. Available: https://arxiv.org/abs/2011.09073

[14] Y. Zha and J. Li, *Virtualizing FPGAs in the Cloud*. New York, NY, USA: Association for Computing Machinery, 2020, p. 845–858. [Online]. Available: https://doi.org/10.1145/3373376.3378491

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

[16] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 220–233.

[17] S. Zeng, G. Dai, H. Sun, J. Liu, S. Li, G. Ge, K. Zhong, K. Guo, Y. Wang, and H. Yang, "A unified FPGA virtualization framework for general-purpose deep neural networks in the cloud," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, dec 2022. [Online]. Available: https://doi.org/10.1145/3480170

[18] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/roscoe pp. 991–1010.

[19] 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*. New York, NY, USA: Association for Computing Machinery, 2020, p. 827–844. [Online]. Available: https://doi.org/10.1145/3373376.3378482

[20] A. Dhar, E. Richter, M. Yu, W. Zuo, X. Wang, N. S. Kim, and D. Chen, "DML: Dynamic partial reconfiguration with scalable task scheduling for multi-applications on FPGAs," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[21] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon, "Reducing FPGA compile time with separate compilation for FPGA building blocks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 153–161.

[22] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507740 p. 933–945.

[23] A. Lalevée, P.-H. Horrein, M. Arzel, M. Hübner, and S. Vaton, "Autoreloc: Automated design flow for bitstream relocation on Xilinx FPGAs," in *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 14–21.

[24] K. Dang Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation Test in Europe Conference Exhibition, 2017*, 2017, pp. 894–897.

[25] S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650–1654, 2009.

[26] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[27] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software-programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[28] Y. Zha and J. Li, "Hetero-ViTAL: A virtualization stack for heterogeneous FPGA clusters," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 470–483.

[29] 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*, ser. APSys '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3124680.3124743

[30] A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, and M. D. Santambrogio, "Resource-efficient scheduling for partially-reconfigurable FPGA-based systems," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 189–197.

[31] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio, "A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015, pp. 1–6.

[32] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, 2009.