DML: Dynamic Partial Reconfiguration With Scalable Task Scheduling for Multi-Applications on FPGAs

Ashutosh Dhar, *Member, IEEE*, Edward Richter, *Student Member, IEEE*, Mang Yu, Wei Zuo, Xiaohao Wang, Nam Sung Kim, *Fellow, IEEE*, and Deming Chen, *Fellow, IEEE*

Abstract—For several new applications, FPGA-based computation has shown better latency and energy efficiency compared to CPU or GPU-based solutions. We note two clear trends in FPGA-based computing. On the edge, the complexity of applications is increasing, requiring more resources than possible on today's edge FPGAs. In contrast, in the data center, FPGA sizes have increased to the point where multiple applications must be mapped to fully utilize the programmable fabric. While these limitations affect two separate domains, they both can be dealt with by using dynamic partial reconfiguration (DPR). Thus, there is a renewed interest to deploy DPR for FPGA-based hardware. In this work, we present *Doing More with Less (DML)* – a methodology for scheduling heterogeneous tasks across an FPGA's resources in a resource efficient manner while effectively hiding the latency of DPR. With the help of an integer linear programming (ILP) based scheduler, we demonstrate the mapping of diverse computational workloads in both cloud and edge-like scenarios. Our novel contributions include: enabling IP-level pipelining and parallelization to exploit the parallelism available within batches of work in our scheduler, and strategies to map and run multiple applications simultaneously. We consider the application of our methodology on real world benchmarks on both small (a Zedboard) and large (a ZCU106) FPGAs, across different workload batching and multiple-application scenarios. Our evaluation proves the real world efficacy of our solution, and we demonstrate an average speedup of 5X and up to 7.65X on a ZCU106 over a bulk-batching baseline via our scheduling strategies. We also demonstrate the scalablity of our scheduler by simultaneously mapping multiple applications to a single FPGA, and explore different approaches to sharing FPGA resources between applications.

Index Terms—Partial reconfiguration, integer linear programming, scheduling, FPGA, dynamic reconfiguration

1 Introduction

FPGAS have proven to be capable of delivering energy-efficient and high-performance accelerators from edge devices to data centers. However, we have observed several challenges toward widespread adoption of FPGAs. First, the complexity of workloads continues to grow rapidly, creating a demand for more FPGA resources. Second, diverse applications may need to be mapped to the same FPGA. Third, systems may require sharing the FPGA's resources between multiple applications or users. Hence, we need a flexible, high performance, and portable solution to enable simultaneous mapping of large and complex workloads on FPGAs. This requirement spans edge and cloud systems, each of which has unique constraints.

Dynamic partial reconfiguration (DPR) presents itself as a strong solution by exploiting the ability to dynamically reconfigure portions of the FPGAs to map workloads in fractions at a time. DPR provides the ability to configure portions of the programmable fabric while other sections continue running. Effectively leveraging DPR, however, is fraught with challenges. First, the latency of DPR can be large, in the order of several milliseconds, adding a significant latency. Second, the design of DPR based FPGA designs is time consuming and difficult, with the designer having to perform manual placement of reconfiguration regions. This makes scaling designs across different FPGAs difficult. Third, DPR designs have statically designed and placed partitions. This makes mapping several workloads to a single large FPGA with DPR to be incredibly laborious. Finally, accelerators often work on batches of data in cloud and edge systems to increase efficiency. Modern DPR solutions must be cognizant of this to exploit all available parallelism and minimize any DPR overheads. Thus, leveraging DPR requires finding a solution to two problems – scheduling and mapping. Scheduling must decide: (1) when to trigger partial reconfiguration and deploy a portion of the workload to the FPGA, and (2) in what order should each fraction of the workload be deployed and executed. The mapping solution decides in what region of the FPGA should each fraction of the workload be placed.

In this work, we present an *end-to-end* solution that considers all sizes of FPGAs, and can perform simultaneous

- *The authors are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA. E-mail: {adhar2, edwardr2, mangyu2, weizuo, xwang165, nskim, dchen}@illinois.edu.*

Manuscript received 31 May 2021; revised 7 Dec. 2021; accepted 14 Dec. 2021. Date of publication 23 Dec. 2021; date of current version 8 Sept. 2022.
This work was supported in part by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - A research collaboration as part of the IBM AI Horizons Network, in part by the Xilinx Center of Excellence, the Xilinx Adaptive Compute Clusters (XACC) program at the University of Illinois Urbana-Champaign, and in part by the NSF CNS under Grant 17-05047. (Corresponding author: Edward Richter.)
Recommended for acceptance by J. Hormigo.
Digital Object Identifier no. 10.1109/TC.2021.3137785

mapping of multiple applications, and parallel pipelines, to the same device. We call our solution *Doing More with Less* (DML). Our work tackles the scalability challenge of traditional DPR solutions and leverages integer linear programming (ILP) based schedulers for optimal simultaneous scheduling and mapping. We combine our offline *static* ILP-based scheduler and mapper with an online *dynamic* runtime in hardware that executes the global DPR order and mapping solution, by combining it with additional dynamic information available at runtime. The scheduler embraces workload batching by enabling task pipelining and unrolls batches to run parallel pipelines. We also present an architecture that partitions the FPGA resources into uniform pieces (called *Slots*), which can be dynamically reconfigured as bespoke accelerator partitions (called *IPs*). This enables accelerator designers to be abstracted away from the limitations of DPR and physical design, enables simplified mapping of multiple workloads to a single FPGA simultaneously, and allows designs to be portable across cloud and edge FPGAs. We further enhance our work by extending our ILP solution to tackle the scheduling and mapping of multiple applications and very large applications across FPGAs of all sizes. We also explore different mapping strategies for sharing FPGA resources between multiple applications via our scheduler. DML uses high quality schedules and mappings from the static ILP-scheduler, is able to minimize design time effort, and leverages dynamic runtime information, and thus provides a balance between usability, scalablity, and performance.

We summarize our contributions as follows:

1) We present DML, an end-to-end DPR methodology. Our solution is comprised of a scalable architecture, based on the constraints of partial reconfiguration, and a novel static ILP-based scheduler and mapper. Our scheduler is capable of pipelining and parallelizing across batch elements, and demonstrates significant performance gains over naive bulk scheduling, and works in concert with a dynamic runtime execution engine.
2) We provide in-depth design space exploration, and explore multiple batching and partitioning schemes. We also evaluate different strategies for simultaneously mapping and scheduling multiple applications to an FPGA.
3) Our solution scales from small to large FPGAs, and we demonstrate the simultaneous mapping of ten applications to a single FPGA via our scheduler.
4) We validate our framework by evaluating real-world benchmarks in hardware, across large and small FPGAs. We demonstrate an average speedup of 5X and up to 7.65X on a ZCU106 via our novel scheduling strategies over naive bulk scheduling.

The rest of this paper is organized as follows. In Section 2 we discuss related and prior work on DPR, followed by an overview of our methodology in Section 3. We then discuss our scheduler, ILP formulation, and mapping strategies for large graphs and multiple applications in Section 4, provide a detailed evaluation in Section 5, and finally conclude in Section 6.

2 Related Work

Using DPR to map large workloads has been explored previously; however, these attempts have focused on a particular application or domain of applications [1], [2]. The use of DPR and task-based scheduling have been explored as well [3], [4], [5], [6], [7], [8], including ILP-based solutions [5], [6], [7], [8]. However, these approaches either focus on (1) a specific optimization or application, or (2) improving the speed and performance of ILP-based scheduling via heuristics. In contrast to our work, they do not consider the constraints and requirements of real-world applications such as the need for pipelining and parallelizing batched workloads, sharing FPGAs between multiple applications, and portability from one FPGA to another.

Prior ILP-based approaches [6], [7], [8] have attempted to mask the latency of reconfiguration by *prefetching configurations*, similar to our work. However, in these works, the authors consider a 2D reconfiguration problem, wherein the FPGA is divided into uniform rows and columns, and tasks can require different amounts of resources. Thus, each workload results in a new floorplan, with each task of the workload occupying a different amount of resources. In contrast to our problem formulation, this approach is based on older Xilinx architectures (Virtex 4 and 5), and is not scalable as it does not consider devices of different sizes, portability, and does not simplify or speedup the DPR-based designs for modern computational workloads. Our solution considers and improves ease of use, portability, performance, and flexibility.

In Deiana *et al.* [5] the authors attempt to perform scheduling and mapping, but the complexity of their ILP limits the scalability, and they demonstrate the use of their iterative solution to schedule up to five tasks at a time only, limiting the efficiency of the solution. A two-stage process was proposed by Purgato *et al.* [4], wherein an additional step is needed to find a feasible floorplan solution. This non-ILP approach cannot guarantee optimality. In contrast, our approach standardizes the search space by using uniform-sized DPR partitions (slots), and simultaneously provides a schedule and mapping solution. Our formulation simplifies the ILP problem, helping find optimal solutions for larger graphs, and scheduling and mapping more tasks at a time.

Multiple recent works have presented frameworks centered around DPR [9], [10]. ReconOS [9] is a framework which extends multithreaded programming abstractions to reconfigurable hardware utilizing DPR. ARTICo [10] is a DPR framework, which provides an automatic toolflow to generate partial bitstreams and a runtime to execute on a number of fixed sized slots. DML differentiates from these works as neither work uses static information to generate a high-performance schedule and mapping to combine with their dynamic runtime. Also, neither framework automatically enables optimizations such as pipelining and parallelism.

Finally, many tools have been proposed to automatically perform stages of the DPR pipeline such as floorplanning [11], [12] and application partitioning [11]. DARTS [11] is a framework which utilizes a mixed ILP formulation to automatically generate a floorplan, partition applications, and generate schedules for real-time applications. In contrast to DML, wherein our objective is to minimize the end-to-end

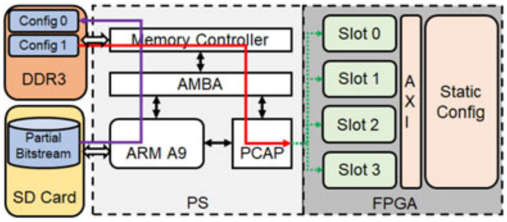


Fig. 1. Proposed system architecture.

latency of a given application or group of applications, DARTS goal is timing predictability and works towards a solution for a user-provided time constraint. Furthermore, DML splits up hardware applications into IPs, which are independently mapped and scheduled with the precedence constraints of the task-graph represented by the constraints of the ILP. DML can then leverage automatic optimizations such as pipelining and parallelism across IPs.

In comparison to prior attempts, our scheduler distinguishes itself with four novel features: (1) Ability to pipeline, (2) Unrolling and parallelizing across batch elements, (3) Graph partitioning optimization to enable the mapping of very large graphs, and (4) Simultaneous scheduling and mapping. In addition, in this work, we do not rely on synthetic graphs. Rather, we consider multiple real-world applications from the Rosetta [13] benchmark suite and validate our framework by testing on real hardware. Finally, we demonstrate the simultaneous scheduling and mapping of multiple applications on a single FPGA and present insights into what strategies work best for such scenarios.

3 Doing More With Less

We now present our *Doing More with Less (DML)* framework – an *end-to-end* and generic methodology that enables any workload, or multiple workloads, to be efficiently mapped to FPGAs of all sizes, with DPR. Our solution is comprised of two key parts. First, we partition the FPGA into uniform pieces, that we call *slots* and provide a scalable architecture, as shown in Fig. 1. Second, we propose an ILP-based optimizer that schedules and maps work into slots, while amortizing the latency of reconfiguration by overlapping the computation with reconfiguration. Our scheduler is capable of pipelining and parallelizing applications by leveraging the data parallelism available across elements in batches of work, and uses a graph partitioning strategy to map very large task graphs. Finally, our flexible architecture and scheduler enable us to simultaneously schedule and map multiple applications on an FPGA.

Leveraging dynamic partial reconfiguration (DPR) requires manual floorplanning to carve out and designate specific regions as static or dynamically reconfigurable. Thus, the designer must decide where to physically place the accelerator. In addition, there are several architectural constraints and design rules that must be considered. A key limiting factor is the speed of DPR, which is determined by the bandwidth available in the *Configuration Access Port* (CAP), and the size of the partial bitstream. The CAP bandwidth is architecture specific and may not be changed, however, the size of the partial bitstream is determined by the size of the dynamically configurable region (DPR Region) and not by how many resources within the DPR region are in use. Note, that while we focus on Xilinx Zynq and Xilinx Zynq Ultrascale+ [14], [15] series FPGA-SoCs in this work, our solution is not limited to Xilinx devices. Fig. 1 presents the system architecture we have designed in this work to help overcome the challenges in leveraging DPR. We designate each slot as a resource partition that includes a reconfigurable partition and a fixed interface. All slots are uniform in their resources and since DPR requires slot interfaces to be uniform, we use AXI-based buses to create their interfaces. The static region of the FPGA hosts the global AXI interconnect, to which the slots connect.

To map an application to our architecture, we partition it at a task level and represent it as a task graph. The task graph is a directed acyclic graph (DAG), $G(V, E)$, such that each vertex, $v_i \in V$, is a task, and each edge, $e_{ij} \in E$, represents a dependency between tasks such that v_i must complete before v_j can begin execution. We then create IPs for each task, and assign the latency of the IP as the weight of the vertex in the task graph. This task graph model is illustrated in Fig. 3a where each vertex represents a task of the application that has its own IP. Edge weights may be used to represent the communication latency. IPs may be designed in any fashion and allow users to deliver fine-grained customization on a per-task level. Alternatively, users may choose to group several tasks or kernels into a single large task and IP. Once the application has been represented as a task graph, it must be scheduled and mapped to slots on the FPGA, which we discuss in the next section. A key advantage of our approach is that by grouping together task graphs of multiple applications, we can create a single task graph. Thus, we can simultaneously schedule multiple applications on a single FPGA, without changes to the applications, floorplan, or scheduler. Note that while DAGs by definition cannot have cycles, DML can address statically resolvable cyclical patterns in the task graph by either unrolling the cycles or absorbing the cycles into a single node.

The size, shape, and location of the slots are determined based on the DPR constraints of the FPGA, and their height spans the entire clock region. This eliminates several of the constraints involved in *2D reconfiguration* described in earlier works [6], [7], [8]. Slot sizes can be set to ensure that the application's IPs are able to fit into them, if the IP library already exists. Note that the size of a slot determines the latency of DPR and limits the performance of an IP that can be mapped into it. Should an application's task require an IP that is too large for a slot size, then we must either split the IP into smaller partitions that map to more than one slot, or we must scale back the IP's performance and reduce its resource requirements. Finally, all slots communicate via AXI in the global address space, i.e., DRAM. Hence, we set the number of slots per FPGA such that the total required DRAM bandwidth does not cause bus contention.

The use of uniform slots is a compromise we make to speedup the design time, scale across all devices, and map multiple applications to the same device simultaneously. In cloud-like scenarios, where multiple applications may need to be mapped, rapid deployment and reducing physical design effort are very valuable. Thus, our architecture provides two key advantages: First, by employing fixed reconfiguration partitions, the designer does not need to perform

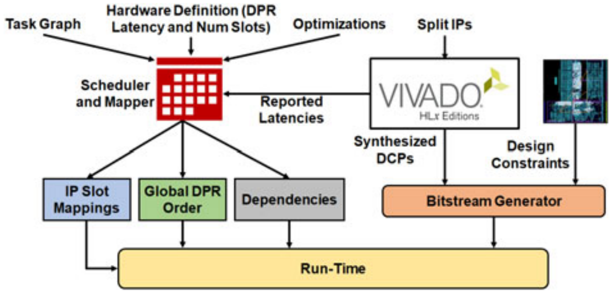


Fig. 2. Doing More with Less methodology and flow.

floor planning for each application, and can design accelerators with defined IO constraints, and be ensured that it will scale across devices. Second, the fixed slot sizes simplify the scheduling and mapping constraints, helping to deliver a scalable and deterministic design with uniform DPR latency. The use of fixed slot sizes may not guarantee the best utilization of the fabric, and requires some thought and planning by the IP designer, as is the case in any IP design effort. However, we believe that the aforementioned benefits far outweigh the utilization benefits of using variable slot sizes. In addition, a different slot size can be selected to better suit the application(s).

Fig. 2 presents the overall flow and framework of our solution. For a given FPGA, we have an overlay architecture that determines the number of slots and DPR latency, and for a given application that needs to be mapped to the FPGA, we have a task graph comprised of kernels. Note that this is *not* a computational overlay, such as a CGRA or a systolic array. Our architecture is flexible and allows us to deliver application and task-specific specialization with high performance. We begin with kernels in the task graph and generate IPs for them via high-level synthesis (HLS).

We then use the reported latency of the IPs, architectural parameters, the chosen level of scheduler optimization (pipelining and parallelism factor), and the task graph as inputs to our static ILP-based scheduler. Note, DML is not suitable for applications with IPs whose latency cannot be estimated prior to runtime. The scheduler then delivers a mapping solution and a DPR and IP execution schedule which can be executed on the hardware by the dynamic runtime. The mapping solution, and DPR and IP execution schedules are represented by three components: (1) global DPR order, which is a list of IPs in the order for them to be reconfigured on the slots (2) IP slot mappings, which map each IP to the physical slot on the device it will be run on, and (3) dependencies, each IP has a list of dependencies extracted from the task graph and used by the runtime. We discuss the operation of this runtime in the next section.

In parallel, we use an automated version of the process described in [16] to generate partial bitstreams, which we call the *Bitstream Generator*. We use synthesized design checkpoints of the IP and our custom overlay floorplan to generate partial bitstreams. We feed the partial bitstreams to a runtime that executes the applications based on the provided mapping and schedule. This runtime is implemented in software and runs on the processing system (*PS*) of the SoC-FPGA. Finally, note that without a slot mapping solution from our scheduler, users would need to design and generate bitstreams for every possible IP-slot pair to enable a *dynamic scheduler* to map *any IP to any slot* at runtime. This adds significant design effort and time overheads.

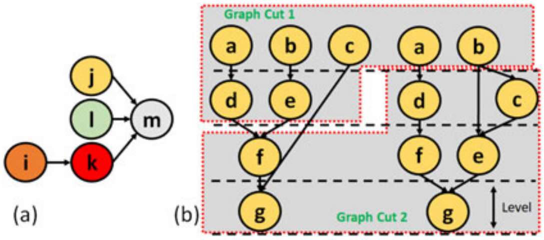


Fig. 3. (a) Example task graph. Each vertex in this graph represents a single task, and the directed edges between vertices represent task dependencies. An IP is created for each vertex, and the weight of each vertex is the latency of its corresponding IP. (b) Sample cut solutions for a large task graph. Each cut has a max size of seven vertices.

4 ILP Based Scheduling

At the heart of the scheduler is our ILP formulation. Our goal is to find a high quality solution, while minimizing the traditional time costs of ILP-based solutions. Our ILP solution performs simultaneous scheduling and mapping and can provide an optimal solution on reasonable graph sizes. Crucially, we consider real-world deployment constraints, and include the ability to pipeline and parallelize tasks across batches. While, our slot-based architecture helps simplify the ILP formulation, finding a solution for the ILP can be slow and does not scale well to large graphs. Hence, we use heuristic schedulers to help tune the ILP solver's search space, and explore different partitioning strategies to find scheduling solutions when trying to map large graphs. We also extend our framework to support two different solutions for mapping multiple applications.

Our ILP solution performs simultaneous scheduling and mapping, can provide an optimal solution on reasonable graph sizes, and takes into account our scalable architecture, which helps loosen the ILP constraints. Crucially, we consider real-world deployment constraints, and include the ability to pipeline and parallelize tasks across batches, and include two different solutions for mapping multiple applications to a single FPGA.

4.1 ILP Formulation

The input to the ILP is an application task-graph, IP latencies, DPR latency, and resource constraints. Our ILP simultaneously looks for a schedule, that provides the global DPR order, and a mapping solution. We formally describe the ILP formulation as follows:

Given: (1) A task graph $G(V, E)$ as described in Section 3; (2) A set of scheduling constraints, C_{S}; and (3) A set of resource constraints, C_{r}. The scheduling constraints include dependencies inferred from the graph, latency of nodes in the graph, and DPR latency. In addition, we must ensure only one partial reconfiguration is done at a time. The resource constraints are the number of available slots, as provided by the user. Additionally, the user may select optimizations, such as pipelining or parallelization, to be included. We will discuss these later in this section.

Goal: Minimize the latency of the entire task graph, such that each task's IP(s) are allocated a slot and experiences the latency of reconfiguration prior to the IP executing in the slot, such that all constraints in C_s and C_r are satisfied.

ILP Variables: We will now define the variables that we will solve to find our solution. We define the set V as all the vertices in the given graph. The sets L and Lpr contain the execution latency of each node in V, and the latency of reconfiguration, respectively. Then, we define the variables S and Spr, as timestamps, where $S \in \mathbb{Z}$ are the start times of all IPs in the set V, and $Spr \in \mathbb{Z}$ are the start times of the corresponding partial reconfigurations of each node in V. Next, we describe our resource mapping variables. Let the number of available slots be R_S. Then, we define a binary variable M_{ik} as $M_{ik} = 1$ if the i-th IP, v_i, maps to the k-th slot, where $v_i \in V$ and $k \in R_s$. Next, since any IP can be mapped to any slot, provided the slot is not occupied, we must express the resource sharing between IPs. We define the binary variable Y_{ijk} as $Y_{ijk} = 1$ if the i-th and j-th IP map to the k-th slot, where $v_i \in V$, $v_j \in V$, and $k \in R_s$. Finally, variables $B1_{ijk}$, $B2_{ijk}$, and $B3_{ijk}$ are Boolean decision variables that we use to help encode our overlap constraints. Their solution is determined by the ILP solver. We also add C_1, C_2, and C_3 as large enough constants, and discuss how to set them later in this section. Next, we describe our system of equations that formulate the constraints of our problem.

Legality Constraints: We encode the fundamental constraints of the system by defining the solution space. We must enforce bounds on start times, ensure that an IP maps to only one slot, and only allow IPs to share a slot one at a time. We begin by enforcing that the start times of all operations must be positive

$$S_i \ > \ 0; \forall \ i \in V \tag{1}$$

$$Spr_i \ > \ 0; \forall \ i \in V. \tag{2}$$

Then, we enforce that an IP can only map to one slot, by using the resource mapping binary variable, M_{ik}, and ensuring only one slot-mapping is set, per IP. Thus we have

$$\sum_k M_{ik} = 1; \forall \ i \in V. \tag{3}$$

Finally, we need to define the resource mapping binary variable Y_{ijk}, that tracks if two IPs use the same slot. Y_{ijk} is key to our ability to constrain and allow IPs and DPR to overlap. Hence, we define Y_{ijk} as $Y_{ijk} = 1$ if and only if, $M_{ik} = 1$ and $M_{jk} = 1, \forall k \in R_s$ and $\forall \ i, j \in V$. We can express this as

$$Y_{ijk} \geq M_{ik} + M_{jk} - 1 \tag{4}$$

$$Y_{ijk} \leq M_{ik} \tag{5}$$

$$Y_{ijk} \leq M_{jk}. \tag{6}$$

Latency and Dependency Constraints: Next, we define our latency and edge dependency constraints. If there is an edge $e_{ij} \in E$, from the i-th IP to the j-th IP, then the start time of the j-th IP must be greater than the sum of the start time of the i-th IP and its latency. Also, an IP can only start once its reconfiguration is complete. Thus, the start time of the i-th IP must be greater than the sum of the start time of the i-th IP's DPR and the latency of reconfiguration. Hence, we add

$$S_j \geq S_i + L_i, \forall e_{i,j} \in E \tag{7}$$

$$S_i \geq Spr_i + Lpr_i, \forall i \in V. \tag{8}$$

Overlap Constraints: Next, we define our overlap constraints that ensure DPR doesn't begin before the previous IP in the slot has completed, only one DPR can be performed at a time, and IPs mapped to the same slot do not try to overlap their execution. Note that in (9) to (14), we use the variables $B1_{ijk}$, $B2_{ijk}$, and $B3_{ijk}$ as a tool to help express an either/or inequality in a way that is amenable to ILP, and C_1, C_2, and C_3 are large enough constants that we set. We further explain and provide insight into these variables later in the section.

Our first constraint is added to enable DPR to overlap with computation. If the i-th and j-th IP map to the same slot, and DPR of the i-th IP takes place before the j-th IP, then the start time of the j-th IP must be greater than the end time of the i-th IP.

Thus for all pairs of i and j, $i \in V$, $j \in V$

$$Spr_i - S_j + C_1.B1_{ijk} \geq L_j.Y_{ijk}, \forall k \in R_s \tag{9}$$

$$Spr_j - S_i + C_1.(1 - B1_{ijk}) \geq L_i.Y_{ijk}, \forall k \in R_s. \tag{10}$$

We must also ensure that if two IPs are mapped to the same slot, they cannot overlap their execution. Thus, if the i-th and j-th IP map to the same slot, the start time of the i-th IP must be greater than the end time of the j-th IP, or vice versa. Hence we have

$$S_i - S_j + C_2.B2_{ijk} \geq L_j.Y_{ijk}, \forall k \in R_s \tag{11}$$

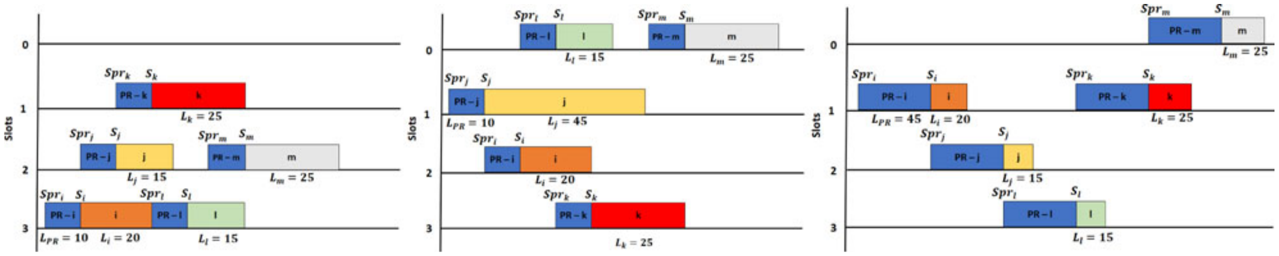
$$S_j - S_i + C_2.(1 - B2_{ijk}) \geq L_i.Y_{ijk}, \forall k \in R_s. \tag{12}$$

Finally, we must ensure that DPRs cannot overlap, since only one DPR can occur at a time. Thus, for all pairs of i-th and j-th IPs, the DPR start time of the i-th IP must be greater than the DPR end time of the j-th IP, and vice versa. We add the constraints

$$Spr_i - Spr_j + C_3.B3_{ijk} \geq Lpr_j, \forall k \in R_s, \forall i, j \in V, i \neq j \tag{13}$$

$$Spr_j - Spr_i + C_3.(1 - B3_{ijk}) \geq Lpr_i, \forall k \in R_s, \forall i, j \in V, i \neq j. \tag{14}$$

As we mentioned earlier, in (9) to (14), we use the variables $B1_{ijk}$, $B2_{ijk}$, and $B3_{ijk}$ as a tool to help express an either/or inequality in a way that is amenable to ILP. These binary variables encode precedence relationships between configuration and execution of IPs i and j. For example, in (9), $B1_{ijk} = 1$ encodes that IP i is configured before IP j is run, while $B1_{ijk} = 0$ encodes IP j is configured before IP i is run. Similarly, $B2_{ijk}$ encodes the precedence relationship between execution times of IP i and j, while $B3_{ijk}$ encodes the precedence relationship between configuration times of IPs i and j. The ILP solver finds a solution for these variables in conjunction with all the other latency, legality, and overlap constraints.



(a) Not all slots are used.

(b) All slots are used.

(c) DPR latency dominates execution.

Fig. 4. Graphical depictions of three different schedules using the task graph in Fig. 3a with varying IP and DPR latencies.

Objective Function: In order to create our objective function, we create a *sink* node, $Vsink$, such that $\forall v \in V$ there exists an edge between Vsink and v, and the start time of the sink node is S_{sink}. Thus our objective function is to minimize the start time of the sink

$$Min(S_{sink}). \tag{15}$$

Leveraging Heuristics: To help express the conditional constraints in Eqs. (9) to (14) we introduced C_1, C_2, and C_3 as large enough constants, and introduced $B1_{ijk}$, $B2_{ijk}$, $B3_{ijk}$ as Boolean decision variables, whose values are determined by the ILP solver. The constants help define the bounds of the solution space and the value of the constant must be very close to the upper bound of the variable. Choosing too small a value might result in an infeasible problem. To help find the bound, we use a heuristic list-scheduler to provide a fast solution. Note that the list-scheduler does not consider mapping solutions, is not optimal, and does not consider all overlap constraints. So, we apply a safety margin to the result of the list-scheduler to determine the bound.

Illustrative Example: To help illustrate the operation of the scheduler, we consider the task graph shown in Fig. 3a and present the resulting mapping and schedules from our scheduler in three different scenarios with different IP and DPR latencies in Fig. 4. For brevity, we restrict our example to four slots, and consider scenarios where: (1) DPR latency dominates execution time (Fig. 4a), (2) DPR latency is easily masked by IP execution latencies (Fig. 4b), and (3) DPR and IP and latencies have varied ratios Fig. 4c. For consistency, we use the same notations as presented in Section 4.1.

Fig. 4 presents the solutions generated by our ILP formulation. The generated solutions illustrate that our *static* ILP-based scheduler will always try to minimize the total execution time. Note that the ILP-based scheduler can find optimal solutions for reasonable problem sizes. However, given that our target objective is to find the best performing order and mapping, it is possible that multiple order-mapping solutions provide the best performance. Thus, the delivered solution may not be intuitive. For example, in Fig. 4a, we can see that the scheduler has opted not to use all four available slots, as it can achieve the minimum latency with just three. This is because IP m is dependent on IPs j, k, and l, and thus i and k form the critical path such that their execution latencies are easily able to mask DPR latency. Moving IP l or IP m into slot 0 would not have improved performance, but it would be *another* solution. Meanwhile, in Fig. 4b, we consider a situation where IP j is on the critical path, but the latency of DPR and the remaining IPs remain the same as in Fig. 4a. Here we can see Eqs. (11)-(12) in action, and they allow IPs i, j, and l to overlap their execution, while Eqs. (9)-(10) helps allow i, k, l, and m to overlap their DPR with IP executions. Note that all four slots are used in this example. Finally, in Fig. 4c, we consider a situation where DPR latencies dominate, and thus Eqs. (13)-(14) are key in enforcing that DPRs do not overlap, while Eqs. (9)-(10) improve performance by allowing DPR and IP execution to overlap.

4.2 Additional Support for Computational Workloads

Batching is commonly used in computational workloads since each kernel needs to be run multiple times for a variety of inputs. It can also help amortize the cost of reconfiguration in some cases. We initially consider batching from three approaches: (1) Bulk batching, wherein a single instance of the IP is re-used for each entry in the batch. For a batch of size N we scale the latency of each IP by a factor of N, thereby serializing the batch but increasing the time each IP must remain configured on the device before it can be swapped for another. (2) Parallel batching, wherein multiple instances can run batch entries in parallel. We replicate the graph N times, thereby creating N parallel instances, which can potentially lead to better slot utilization but blows up the size of the ILP problem, making it harder to find a solution. Note that both of these approaches are possible with our described ILP formulation. (3) Pipelining across batches, wherein like bulk batching, each IP's latency is scaled by a factor of N; however, we allow dependent IP to overlap their execution since the dependencies exist within a batch entry only. Thus, each pipeline stage is an IP, operating on a separate entry in the batch. In order to do so, we extend our ILP formulation. If there is an edge $e_{ij} \in E$, from the i-th IP to the j-th IP, then the start time of the j-th IP must be greater than the sum of the start time of the i-th IP and the latency of one entry in the batch. Also, the penultimate batch of the j-th IP must begin only after the last batch of the i-th IP has completed. Thus we have

$$S_j \geq S_i + L_i/N; \forall\, i, j \in V \tag{16}$$

$$S_j + (N-1).L_j/N \geq S_i + L_i; \forall\, i, j \in V. \tag{17}$$

Note that we assume that the latency of the i-th IP, L_i has already been scaled by a factor of N. However, should the latency of the j-th IP be much smaller than the i-th IP, we must ensure that it still completes after the i-th IP and respects the dependency

$$S_j + L_j \geq S_i + L_i; \forall\, i, j \in V. \tag{18}$$

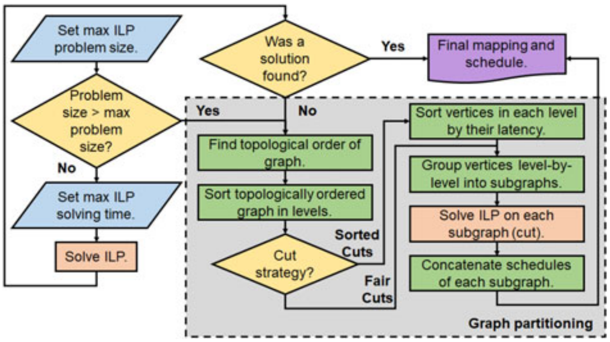

Set max ILP problem size.
Problem size > max problem size?
No
Yes
Set max ILP solving time.
Solve ILP.
Was a solution found?
Yes
No
Final mapping and schedule.
Find topological order of graph.
Sort topologically ordered graph in levels.
Cut strategy?
Sorted Cuts
Fair Cuts
Sort vertices in each level by their latency.
Group vertices level-by-level into subgraphs.
Solve ILP on each subgraph (cut).
Concatenate schedules of each subgraph.
Graph partitioning

Fig. 5. DML ILP scheduler and graph partitioning flow.

4.3 Large Graphs and Multiple Applications

ILP based solvers can be slow and do not scale to large graphs. This can be challenging when we attempt to map very large applications, leverage parallelism (as it duplicates the graph), or map multiple applications on to a single FPGA. In this section we discuss our approach to dealing with very large task graphs and multiple applications.

Handling Large Graphs: While ILP solvers are slow, it is possible to find a non-optimal solution for the entire task graph or find optimal solutions for subgraphs within reasonable time. Thus, DML uses graph partitioning to schedule large graphs. This is applicable for applications with very large task graphs, leveraging parallel batching for a single application, or while trying to map multiple applications to a single FPGA. We illustrate our overall ILP-based scheduler flow in Fig. 5. We begin by setting upper bounds on the size of the task graph and ILP solving time. If the task graph is too big, or a solution cannot be found within the time period, we partition the graph.

DML partitions the task graph into smaller subgraphs, called *cuts*, performs ILP-based scheduling of each cut, and then sequentially concatenates the schedules and mappings of each cut to create the final global schedule and mapping. As shown in Fig. 5, the graph partitioning begins by performing topological sorting of the graph and sorts vertices into levels, such that vertices in a level are tasks/IPs that have no dependence on each other but are dependent on vertices in higher levels. We then group together vertices, level-by-level, into *cuts* of fixed sizes. When selecting vertices to be placed into a cut, DML uses two different strategies: (1) sorted cuts, and (2) fair cuts. The sorted cut strategy considers scenarios where we are trying to schedule and map multiple applications to a single FPGA at the same time, where the task graph might be very large and have tasks/IPs with very different execution latencies. Grouping together vertices into a cut without considering the different execution times may result in a final schedule with large bubbles in the pipeline. Thus, in the *sorted cut* strategy, DML will first sort the vertices in a level by their latencies. Thus, when cuts are formed, we are less likely to have vertices with vastly different execution times, thereby improving FPGA utilization. In contrast, The *fair cut* strategy considers the case of parallel batching, where we could have a very large graph, such that the latency of nodes in each level are identical to each other. In this case, we can schedule all nodes without bias. Having formed the cuts, we then schedule and execute each cut sequentially. This scheduling algorithm illustrated in the flow chart in Fig. 5, while Fig. 3b illustrates cuts and levels in a large task graph.

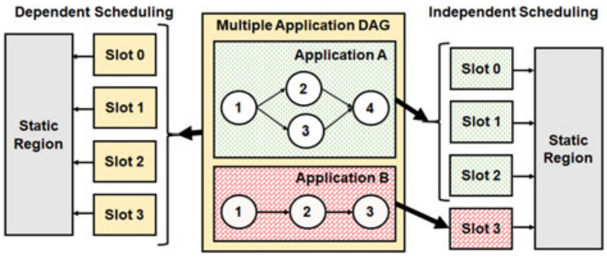


Fig. 6. Illustration of dependent versus independent scheduling of multiple applications.

Strategies for Multiple Applications: Without DPR, running multiple applications on an FPGA would require time multiplexing the applications over the entire FPGA, which would be slow and would limit the ability to exploit the available fine-grained customization and parallelism. We call this approach *Coarse-Grained Scheduling*. As we discussed in Section 3, our flexible architecture and scheduler enables us to leverage DPR to share an FPGA across multiple applications simultaneously. We combine the task graphs of all applications into a single monolithic graph, and our scheduler finds a solution for all applications simultaneously, as we discussed in Section 4.3. We refer to this method as *dependent-scheduling*, and illustrate it in Fig. 6. Note that any IP of any application is free to be mapped to any slot of the FPGA. This method can efficiently use all FPGA resources and infrastructure, and can attempt to provide the optimal end-to-end latency for the monolithic DAG. However, there are a few potential disadvantages of this approach: (1) It only attempts to optimize the total latency, not the per-application latency, (2) The monolithic DAG can be very large which requires graph cutting to generate a schedule in a timely fashion, which will create a less-performant schedule, and (3) As the graph is presented as a monolithic DAG, schedule generation can be slow as only one ILP solver is run.

Thus, in this work, we consider an alternative approach to multiple applications - *independent-scheduling*. We statically designate a number of slots to each application, and then independently run the scheduler on each application for their designated number of slots. Thus, each application runs with its best performance and does not interfere with scheduling and ordering or other applications. This, however, does come at the cost of a potentially longer end-to-end latency for the entire group of applications. We illustrate this scheme on the right side of Fig. 6. Note, however, that the FPGA still has a single CAP interface that must be shared across all applications. Our runtime allocates this in a round-robin fashion to each application.

Our DML framework is flexible and can implement either multiple application mapping strategy based on the optimization goals. We explore the difference between dependent and independent multi-app scheduling in Section 5, and demonstrate scenarios where each is beneficial. Finally, we note that another advantage of independent scheduling is the ability to tune the number of slots based on application characteristics. While intuitively one may think that the number of slots required is proportional to the number of IPs in an application's task graph, our findings show that the number of slots needed depends on the topology of the task graph, the latency of the IPs, and the batch size used. Section 5.7 provides a quantitative analysis and demonstrates the benefit of providing a bespoke number of slots to each application.

4.4 Runtime Implementation

The DML runtime executes the schedule generated by the scheduler. However, the static ILP scheduler is fed simulated latencies of the IPs, which can differ from the actual hardware latency. Thus, instead of using the *exact* IP start times predicted by the scheduler, DML's dynamic runtime uses the mapping and global DPR order generated by the ILP-based scheduler, along with the dependencies of the task graph to execute the application. During execution, the runtime iterates over the global DPR order. When the slot of the next IP in the global DPR order is available, and no previous DPR is running, the runtime will start DPR in the slot denoted by the mapping. While waiting for the next DPR, the runtime iterates over each IP that has already been configured, and checks if the dependencies are complete. If the dependencies are complete, the runtime will start the execution of the IP. This approach is beneficial as it combines the static high-order information from the ILP-based scheduler, such as mapping and global DPR order, with the dynamic information the runtime has such as the exact time DPR is available, or an application's dependencies are complete. While the operation of the dynamic runtime makes it possible that the IP start time in hardware differs from that predicted by the static scheduler, this does not impact performance. We are simply adjusting DPR or IP start times by using available slack in the schedule, between IP execution ending and DPR beginning in a slot, while still following the global DPR order, IP slot mapping, and dependencies. We quantitatively show in Section 5.3.1 that the acquired speedup in the hardware matches or outperforms that estimated by the scheduler, which is further explained in our evaluation.

5 Evaluation

We now present an evaluation and exploration of our DML framework. As we discussed in Section 3, our architecture uses slots of uniform resources and interfaces. In this work, our slots include 10000 LUTs, 40 DSPs, and 40 BRAM18 units. We target two different sizes and architectures of FPGAs: (1) A Xilinx Zynq-7000 based Zedboard, and (2) A Xilinx Zynq Ultrascale+ ZCU106 board. Slots on the Zedboard result in partial bitstreams of 1.2MiB that take 9.5ms to reconfigure, while the ZCU106 slots are 0.98MiB in size and take 2.9ms to reconfigure. This amounts to an average reconfiguration bandwidth of 125 MiB/s and 340 MiB/s for the Zedboard and ZCU106 respectively. We refer to these as 1X sized slots, and consider slots of twice as many resources as well, and call them 2X slots.

We consider real-world benchmarks, as provided by the Rosetta [13] benchmark suite – 3D Rendering (3DR), Digit Recognition (DR), and Optical Flow (OF) – and in-house developed accelerators for Alexnet (AL4) and LeNet (LN) neural networks, and Image Compression (IMGC). All benchmarks were developed with Xilinx HLS tools, and we modified the benchmarks by splitting them up into smaller task modules and generating uniform AXI interfaces (via HLS pragmas) for the IPs. We attempt to get the best possible performance from the available slot resources, and where possible we split the task IPs across multiple slots, especially in data-parallel applications like the DNNs (AL4 has four parallel branches). We also consider four synthetic graphs, similar to those in Fig. 3b, to add diverse patterns to our multi-application studies.

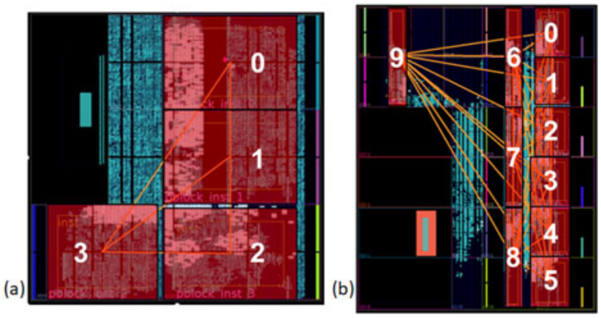


Fig. 7. Floorplans used to evaluate DML on (a) Zedboard and (b) Zynq Ultrascale+ ZCU106, for 1X size slots.

5.1 Methodology

Our scheduler is written in Python and uses CVXPY [17] and Gurobi 8.1 [18] for the ILP backend. Our experiments are run on a cluster with Intel Xeon E5-2680 v4, and we restrict Gurobi to use four threads only.[1] ILP latencies of the application's task IPs have been generated from Xilinx Vivado HLS synthesis and co-simulation. In this section, we present data generated by our scheduler to perform a detailed sweep, design space explorations, and to prove the scalability of the methodology. We present data for both, the Zedboard and the ZCU106, as they have different architectures and DPR latencies. We also provide hardware validation of our methodology by running the generated schedules on the Zedboard and the ZCU106. We performed manual floorplanning on both devices, to carve out equal-sized programmable regions. We were able to fit four 1X sized slots on the Zedboard, and ten 1X sized slots on the ZCU106 board. Fig. 7 shows these two floorplans, with labeled red rectangles denoting the slots.

Fig. 7b shows the 10-slot floorplan used on the ZCU106. We can see that the static region, which hosts the AXI interconnects, is placed in the middle of the board, near the PS-PL interface, and with the programmable slots on the outer edges of the device. We also note the different slot aspect ratios on the ZCU106 floorplan (i.e., Slot 6 is taller and thinner than Slot 0). This is necessary as Xilinx FPGAs are not uniform in their placement of DSP and BRAM columns. Hence, some slots must be taller and thinner to consume the correct number of BRAMs and DSPs. We did not find that this difference in aspect ratio impacted the timing of the IPs within the slots. Note that we attempted to fit 12 1X sized slots on the ZCU106, but the additional slots created routing congestion and we could not meet timing.

[1] To find a solution in reasonable time, we limit solver time to between 600 and 720 seconds. We empirically determined task graphs of 25 vertices to be the upper bound that the ILP could attempt to solve before we require partitioning, as described in Section 4.3. The partitioner uses cuts of size 10 to 15, and defaults to fair-cut method.

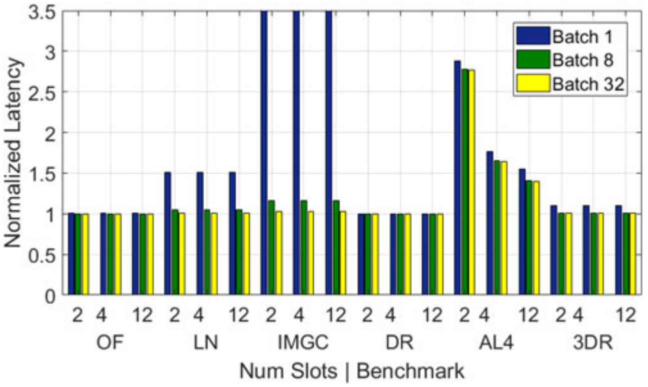


Fig. 8. Normalized end-to-end latency of applications across batch sizes and number of available slots, for 1X sized slots on a Zynq-7000 device. For ease of presentation, IMGC batch size-1 has been cut-off, and extends to 6.2X.

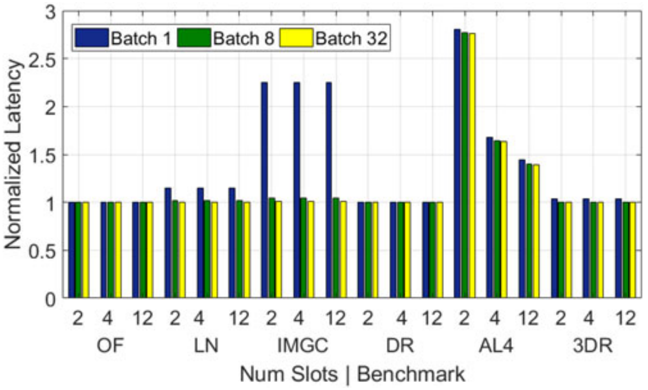


Fig. 9. Normalized end-to-end latency of applications across batch sizes and number of available slots, for 1X sized slots on a Zynq Ultrascale+.

The runtime is run on the baremetal platform provided by Xilinx. We utilize APIs available in the Board Support Package (BSP) to configure slots via the Processor Configuration Access Port (PCAP), which is the CAP connected to the PS on the SoC. The global DPR order, IP slot mappings, and dependencies are generated by the ILP static scheduler as global array definitions in a header file, which is read by the runtime when executing the application(s).

5.2 Exploring the Impact of DPR on Performance

We begin by exploring the impact of batch size and number of slots. We first present the application's end-to-end latency, as reported by the scheduler, normalized against the end-to-end latency of the application without DPR overheads (baseline). For now, we consider bulk batching. Figs. 8 and 9 present the normalized latency of applications, when mapped to 1X sized slots, as mapped to a Zynq-7000 (Zedboard) and a Zynq Ultrascale+ (ZCU106) device. Applications that demonstrate a normalized latency of 1.0 are operating completely unperturbed by DPR overheads. In this study, we sweep the size of the batch, as well as the number of available slots. These results use HLS estimated latencies and measured DPR time. The HLS estimated latencies for all 6 applications range from 7.18ms to 1.91 minutes. The DPR time of a 1X slot on the Zedboard and ZCU106 is 9.5ms and 2.9ms respectively. This is why Figs. 8 and 9 show such diversity in the performance impact of DPR.

As we can see, batch size can have a significant impact on the effective latency as it helps amortize the cost of DPR. For a batch size of 1, many applications are unable to effectively mask the DPR latency. This is very clear in LN and IMGC, where the latency of DPR can be greater than that of the IPs or the application itself. In the case of AL4, the size of the application requires multiple PRs to be done, thus incurring more overhead. However, as the batch size increases, we see almost all applications are able to mask the latency of DPR. This matches our expectation and confirms that the scheduler is performing as expected. Alexnet (AL4) performs poorly even at large batch sizes, when the number of slots available is just two. This is because its task graph is large and has many parallel branches. However, due to the large amount of task parallelism available in AL4, it is able to effectively utilize the additional slots. The remaining apps do not benefit much beyond two slots when we apply bulk batching, due to limited task parallelism. Finally, Figs. 8 and 9 show that the reduced DPR latency on the Zynq Ultrascale+ provides a significant reduction in overhead Overall, with the exception of AlexNet (AL4), we were able to match the baseline performance, despite DPR overheads, given a large enough batch or a sufficient number of slots with bulk batching alone.

5.3 Exploring Batching Strategies

Next, we consider the performance of our generated schedules across different batching strategies. For the sake of brevity, we will not sweep across the number of slots, and will consider the performance of the Zedboard with its four slots, and the ZCU106 with its ten slots. We consider batch sizes of 4 and 32, and explore five scenarios – Bulk batching, Pipelining only, Pipelining and four-way parallel batching (Parallel 4X), Pipelining and eight-way parallel batching (Parallel 8X), and No DPR. No DPR is a hypothetical scenario where we do not use DPR. Instead, we statically fill 80% of the FPGA with as many copies as possible of the entire task graph to mimic a data-parallel approach to computing. We assume 20% of the FPGA's resources are needed for units like AXI crossbars, memory controllers, etc. Figs. 10 and 11 presents the end-to-end application speedup over the baseline, which assumes bulk batching as well but no DPR.

Enabling pipelining allows the scheduler to effectively overlap multiple IP executions. The amount of overlapping is determined by DPR latency, the DAG topology, the number of slots available, and the batch size. Thus, we see that pipelining alone is able to speedup execution by up to 2.2X and 2.73X on the Zedboard and ZCU106, respectively for a batch size of 32. On an average, we see speedups of 1.5X and 1.8X on the Zed and ZCU106, respectively. Note that for small batch sizes, pipelining is unable to provide speedups for applications like IMGC, where the latency of DPR dominates execution.

By enabling parallelization, we unlock even more opportunities for the scheduler. Note, however, the lack of available slots limits its ability to fully exploit the parallelism, and the increased task graph size forces us to use graph partitioning, which further limits the performance of the schedules. As we can see on the Zedboard, parallelism provides up to 3.9X speedups, with an average of 2.67X. Note that the performance of eight-way parallel batching is poor, due to

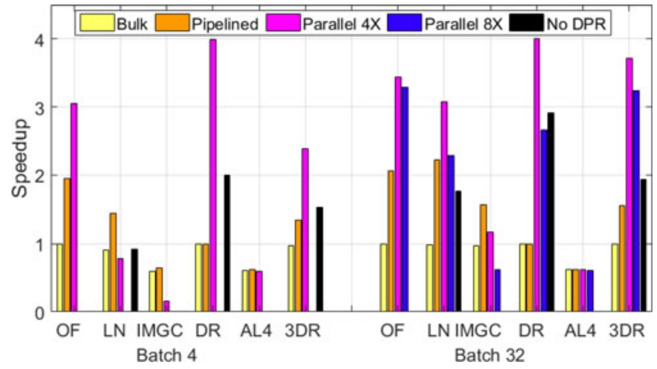


Fig. 10. Impact of batching strategies. Relative speedup shown for batch size 4 and 32 on a Zedboard, as predicted by the scheduler. No DPR solutions cannot fit a single instance of IMGC, OF, and AL4 in a Zedboard. Note: 8-way parallel batching cannot be done on batch of 4.

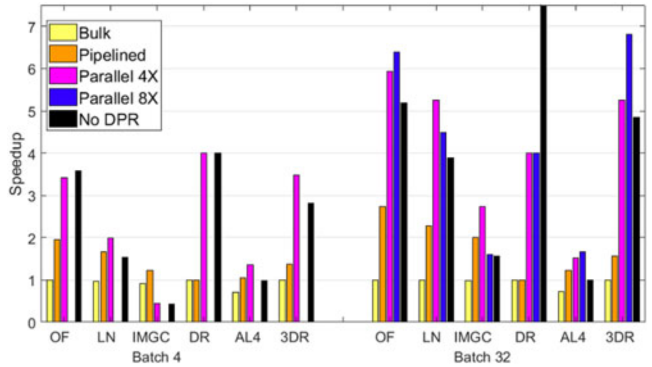


Fig. 11. Impact of batching strategies. Relative speedup shown for batch size 4 and 32 on a ZCU106, as predicted by the scheduler. DR on batch of 32 in No DPR scenario is clipped for presentation, and it extends to 31.8X. Note: 8-way parallel batching cannot be done on batch of 4.

the lack of available slots and graph partitioning. In contrast, we see up to 6.8X speedup on the ZCU106 with the help of eight-way parallel batching, and 4.15X on average. Thus, our scheduler is able to effectively utilize the available FPGA resources and parallelism in the graphs and batches.

We also note that the relatively limited resources of the Zedboard does not allow many of the applications to map to it. Thus, we see that the *No DPR* solution was unable to provide a solution for OF, IMGC, and AL4. This further highlights the need for our DML strategy, which allows compute to be mapped efficiently to any device. In cases where the applications do fit, our pipelining and parallelization approach is able to perform better by better utilization of the FPGA resources. On the ZCU106, which has significantly more resources, we see that for small batch sizes, it might be advantageous to simply instantiate copies of the application (No DPR) if the application is very small, such as DR. However, given enough parallelism, we see that our scheduler is still able to better utilize the FPGA resources, even on the ZCU106.

Next, we examine the effective utilization of resources by our methodology by considering the *slot utilization*. Effective slot utilization measures what percentage of available slots were used over the run of the application on an average. A utilization of 100% would imply all slots were used across the run. Figs. 12 and 13 present our analysis.

Once again we see the effectiveness of our scheduling solutions. On the Zedboard, pipelining alone is able to keep our utilization at 50% on average, while enabling parallelization brings it up to 84% on average, and up to 99%. The ZCU106 has more resources, and can be harder to keep busy for small applications and batch sizes. However, with the help of parallelization, we are able to effectively utilize 53% on average, and up to 88%. Note that parallelization approach forces the scheduler to partition and perform localized scheduling which limits the efficiency.

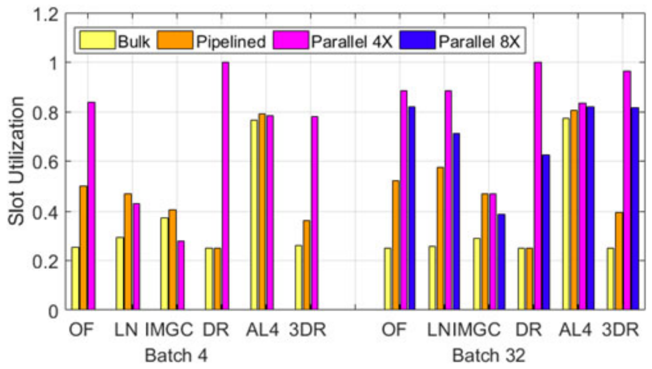


Fig. 12. Impact of different batching strategies. Slot utilization shown for batch size 4 and 32 on a Zedboard, as predicted by the scheduler.

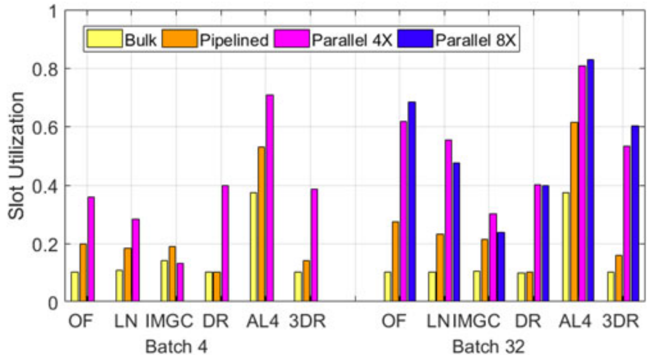


Fig. 13. Impact of different batching strategies. Slot utilization shown for batch size 4 and 32 on a ZCU106, as predicted by the scheduler.

5.3.1 HW Evaluation

Having demonstrated the efficacy of our scheduler and its different batching strategies, we will now evaluate them on real systems. Figs. 14 and 15 show the speedup with the same baseline for bulk-batching, pipelining, four-way parallel batching, and eight-way parallel batching on the Zedboard and ZCU106 respectively. Once again, we observe that pipelining and parallelism can greatly increase the performance of partial reconfiguration applications on our hardware implementation. On the Zedboard for batch 32 we observe an average speedup of 2.87X across all applications and a max speedup of 3.98X. On the ZCU106 for batch 32, we observe an average speedup of 4.99X and a max speedup of 7.65X. Here we observe that the speedup seen in the hardware implementation is higher than that predicted by the scheduler. For a batch size of 32, the average speedup across all applications as measured on hardware is 1.04X and 1.12X higher than that predicted by the scheduler for the Zedboard and ZCU106 respectively. This can be attributed to the IPs running slower in hardware than predicted by Vivado HLS. Thus, DPR latency

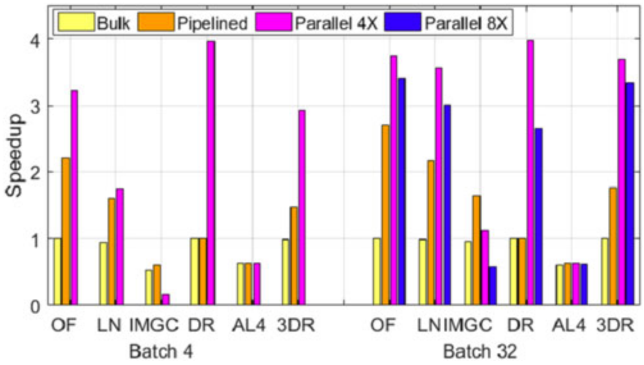


Fig. 14. Impact of different batching strategies. Relative speedup shown for batch size 4 and 32 on a Zedboard, as measured on hardware. Note: 8-way parallel batching cannot be done on batch of 4.

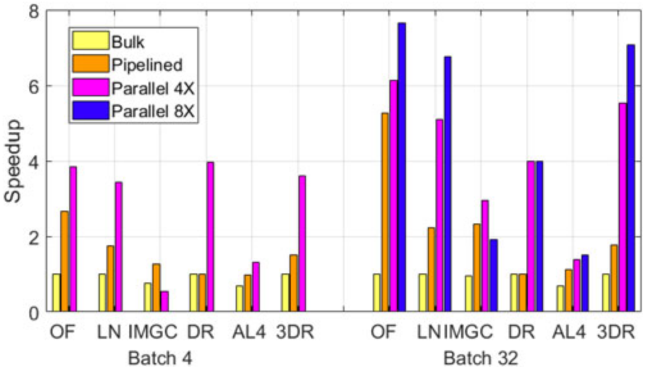


Fig. 15. Impact of different batching strategies. Relative speedup shown for batch size 4 and 32 on a ZCU106, as measured on hardware. Note: 8-way parallel batching cannot be done on batch of 4.

is shorter, relative to the total runtime of the IP, in hardware. This reduces the relative overhead of DPR, making it easier to hide. Hence, the latency predicted by the scheduler is longer, which results in the predicted speedup being more pessimistic. In the case of our parallelization strategies, which provide the best performance, frequent PRs must be performed, and thus the *average* speedup predicted by the scheduler is slightly lower than what we observe in hardware.

Comparing Figs. 14 and 15 with Figs. 10 and 11 shows that the performance trends are the same in both the scheduler and the hardware. In all but two instances, the best performing optimization, as predicted by the scheduler, is the best optimization in hardware, on all boards and batch sizes. This means that the scheduler's performance model is able to effectively model the performance of the hardware implementation. The two discrepancies are LeNet with a batch of 4 on the Zedboard, and LeNet with a batch of 32 on the ZCU106. This is because LeNet contains small IPs which have a very low latency, making it more difficult to hide the latency of DPR. As mentioned previously, the cost of DPR in the scheduler's performance model is higher than that in the hardware. As LeNet already has difficulty masking the latency of DPR, in both discrepencies, the scheduler predicts a solution with less DPR (pipelining with batch 4 and four-way parallel batching with batch 32) would be more performant. However, we see in the hardware that LeNet is actually able to hide the DPR latency, and is most performant with the maximum amount of parallelism available for either batch.

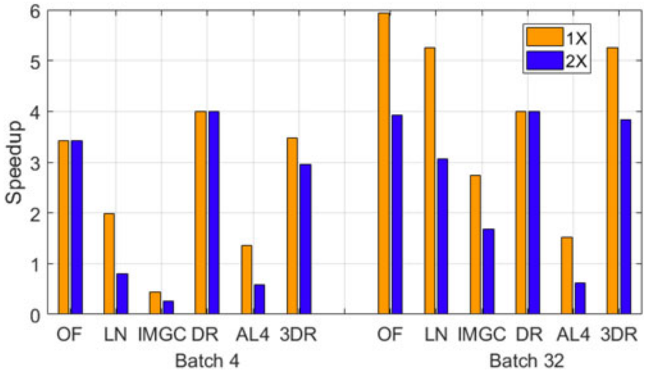


Fig. 16. Impact of slot size on performance. Relative speedup shown for batch size 4 and 32 on a ZCU106 with 4-way parallel batching, for slot size 1X and 2X.

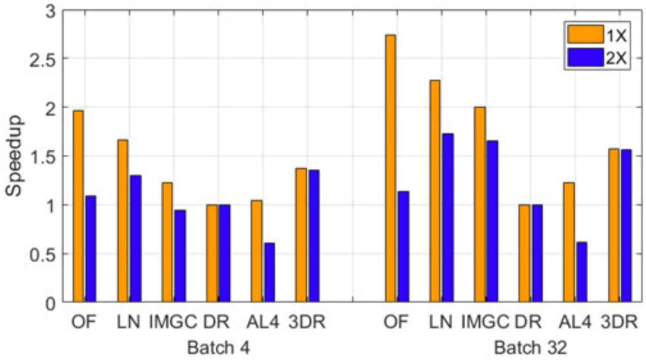


Fig. 17. Impact of slot size on performance. Relative speedup shown for batch of 4 and 32 on a ZCU106 with pipelining, for slot size 1X and 2X.

5.4 Choosing Slot Size

So far, in this section we have only considered 1X sized slots. We will now explore the impact of choosing a larger slot size. Figs. 16 and 17 show the speedups for both 1X and 2X slot sizes on the ZCU106 when performing pipelining and four-way parallel batching respectively. The 1X slot design is using ten slots while the 2X design is using four slots. As one can see, using 1X slots always achieves a higher or the same speedup when compared to the 2X slots. This is for several reasons. First, not all IPs are able to fill the 2X slot, wasting precious resources. Second, 1X gives more flexibility for how IPs can be mapped across slots in time and space, as there are more IPs and more slots. This gives the ILP scheduler a larger space to find a high-performance schedule. Third, 2X slots take almost twice as long to reconfigure, thereby increasing the impact of DPR latency. Finally, 1X slots give more fine-grained specialization than 2X, allowing each IP to be more specialized to the specific computation it is performing.

5.5 Scalability and Multiple Applications

We now demonstrate the scalability of our solution, and used our scheduler to simultaneously map ten applications (The six application previously mentioned plus four synthetic benchmarks) to a single FPGA, across varying batch, resource, and slot sizes, and evaluate our fair-cut and sorted-cut methods. Table 1 presents the scale of the problem, for a batch size of 32, with eight available slots, with pipelining enabled. Since our scheduler runs ILP on several

TABLE 1
Problem Size for Mapping Multiple Applications

SlotSize	Time(s)	ILP(s)	Node	MaxVar	MinVar	AvgVar
1X	768.3	23.4	178	3735	2808	3657.5
2X	318.9	22.1	83	3735	1068	3290.5

cuts of the graph, we present the average number of ILP variables that are solved, along with the max and min. We also list the total time taken to find the final solution, as well as the total time spent solving the ILP alone.

Table 2 presents the normalized speedup of mapping multiple applications on to a single FPGA, as predicted by the scheduler. We consider FPGAs ranging from a small edge-scale device with four slots, to a large cloud-scale with sixteen slots, and consider slot sizes of 1X and 2X, batch sizes of 4 to 32. Here we consider coarse-grained multi-app scheduling, as discussed in Section 4.3 to be our baseline, wherein each application executes serially, incurring one time reconfiguration cost for each application, and assume that there are enough resources for the entire application to fit. Using this baseline, we present the speedup, as reported by our scheduler for the sorted-cut and fair-cut schemes.

TABLE 2
Mapping Multiple Applications

		1X		2X	
BatchSize	Slots	Fair	Sorted	Fair	Sorted
		Speedup			
4	4	0.93x	0.93x	1.51x	1.49x
	8	1.16x	1.15x	2.14x	2.36x
	10	1.21x	1.15x	2.48x	2.44x
	16	1.22x	1.19x	2.57x	2.49x
16	4	1.15x	1.15x	1.51x	1.50x
	8	1.47x	1.43x	2.13x	2.42x
	10	1.52x	1.44x	2.48x	2.52x
	16	1.52x	1.49x	2.55x	2.53x
32	4	1.16x	1.16x	1.51x	1.50x
	8	1.48x	1.43x	2.13x	2.43x
	10	1.53x	1.45x	2.48x	2.53x
	16	1.53x	1.50x	2.55x	2.54x
		Slot Utilization			
4	4	0.64	0.63	0.89	0.86
	8	0.41	0.41	0.61	0.67
	10	0.34	0.33	0.59	0.56
	16	0.21	0.21	0.38	0.35
16	4	0.66	0.64	0.88	0.86
	8	0.43	0.43	0.6	0.68
	10	0.37	0.35	0.59	0.57
	16	0.23	0.23	0.38	0.36
32	4	0.66	0.64	0.88	0.86
	8	0.44	0.43	0.60	0.68
	10	0.37	0.35	0.59	0.58
	16	0.23	0.23	0.39	0.36

TABLE 3
Speedups on Hardware and Scheduler Over Coarse-Grained Multi-App Scheduling for Four Apps

	Fair		Sorted	
Batch	Sched	HW	Sched	HW
4	1.30x	1.34x	1.44x	1.28x
16	1.40x	1.30x	1.64x	1.48x
32	1.42x	Out-of-Memory	1.68x	Out-of-Memory

We observe that our schedule is faster than the baseline across almost every case. This is in part due to our scheduler's ability to pipeline and overlap the execution of IPs, even within smaller graph cuts. Here we note that the sorted and fair cut strategies perform similarly. In this group of applications, a few select applications dominate the end-to-end runtime. Thus, no matter how we perform the graph cuts, the critical path is determined by the same set of vertices. Also, we can see that for larger batch sizes and more slots, the effective slot utilization is poor. This is due to the large disparity in task graphs. Larger graphs, with long latencies and limited task-parallelism, consume the tail end of the schedule, and only require one or two slots.

Once again, we will validate our scheduler by testing it in hardware. We consider four applications, OF, LN, AL4, and 3DR, running simultaneously on the ZCU106, with ten 1X slots. We present the end-to-end speedup as measured on the board versus the scheduler prediction in Table 3. As we can see, the hardware performance once again matches the predicted scheduler performance. Note that the ZCU106 did not have sufficient memory (off-chip DRAM) to host all four applications with a batch size of 32. In addition, unlike our 10 application experiment, the sorted cut strategy outperforms the fair-cut strategy as the end-to-end latency is not dominated by a single application in the tail-end.

5.6 Dependent versus Independent Scheduling for Multiple Applications

We will now explore the impact of different scheduling approaches for multiple applications. As discussed in Section 4.3, the DML framework allows for two multi-application scheduling schemes: *independent-scheduling* and *dependent-scheduling*. We use both scheduling schemes to generate multi-app schedules for four concurrently running applications: LeNet, AlexNet, Optical Flow, and 3D Rendering, and consider the total end-to-end latency of all four applications, and the end-to-end latency of each application. We run our experiments on the ZCU106 board with ten 1X slots, and a batch size of 16. For independent scheduling, we consider three different allocations of slots per application. As discussed in Section 4.3, in dependent scheduling, the scheduler will allocate slots to IPs from the global pool and try to optimize the total end-to-end latency.

Fig. 18 shows the end-to-end speedup of the independent-schedule over the dependent-schedule for the total runtime of all application and the per-application runtime. The per-application slot allocations are shown in the legend. Independent scheduling always has worse end-to-end latency, and on average increases the end-to-end latency by 1.74X. This is because the dependent scheduling optimizes over a monolithic graph across all available slots, so it can prioritize the applications with the longest latency. However, dependent scheduling only optimizes the end-to-end

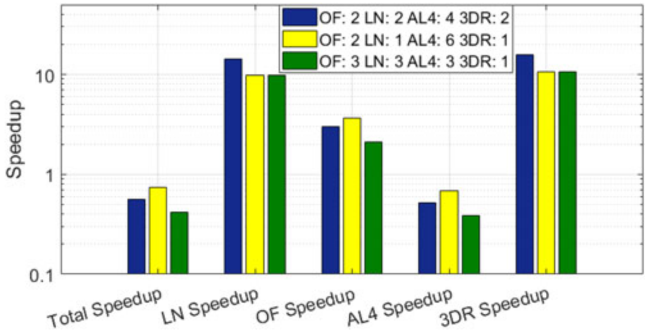


Fig. 18. Speedup of independent over dependent scheduling for total end-to-end latency and on a per-application basis. We consider a mix of four application, with a batch size of 16, and measure performance on a ZCU106 in hardware. Legend denotes the number of slots provided to each application as follows: *NAME: SLOTS*.

latency, which can severely hurt single application performance by unfairly providing slots to the slowest application. Fig. 18 shows that when running four applications, providing two dedicated slots to LeNet gives a speedup of 14.4X, and providing a single dedicated slot to LeNet gives a speedup of 9.8X over if they were dependently scheduled. This can also be seen in 3DR where independently scheduling provides a speedup of 15.7X and 10.7X for two and one dedicated slots respectively. These speedups are so large as depdendent scheduling will prioritize applications with the longest latency, which is AlexNet. AlexNet has 38 IPs which is much more than Optical Flow's nine IPs, LeNet's three IPs, or 3DR's three IPs. AlexNet also has parallel branches in its DAG, allowing it to consume many slots at the same time to further increase performance. Thus, the dependent scheduler will give a large majority of the slots to AlexNet to reduce its end-to-end latency as much as possible. While this is good for end-to-end latency as well as the latency of AlexNet, one can see it unfairly hurts the performance of other applications. In this case, it is beneficial to use independent scheduling to share the slots across the four applications.

In contrast, we will now consider cases where dependent scheduling is better. Fig. 19 shows the speedup that independent scheduling has over dependent scheduling when scheduling three applications: LeNet, Optical Flow, and 3DR. In this scenario, we see that independent scheduling actually hinders the performance. This is due to two reasons: First, these three applications consist of 15 IPs, which is small enough for the dependent scheduling to find an optimal solution, without partitioning, for the entire DAG. Second, these applications have very limited branches, and do not require a larger number of slots for maximum performance. In this scenario, there is less contention for slots, which results in improved per-application performance. Thus, it is better to use dependent scheduling.

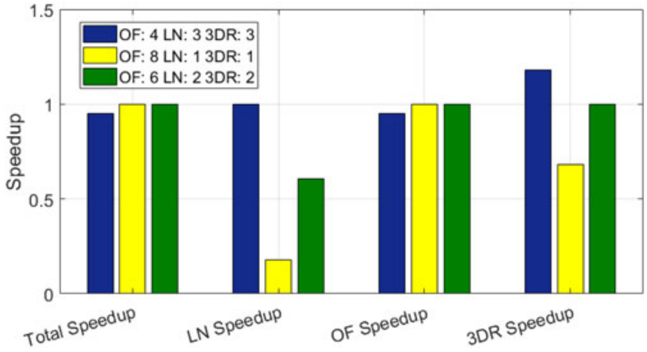


Fig. 19. Speedup of independent over dependent scheduling for total end-to-end latency and on a per-application basis. We consider a mix of three application, with a batch size of 16, and measure performance on a ZCU106 in hardware. Legend denotes the number of slots provided to each application as follows: *NAME: SLOTS*.

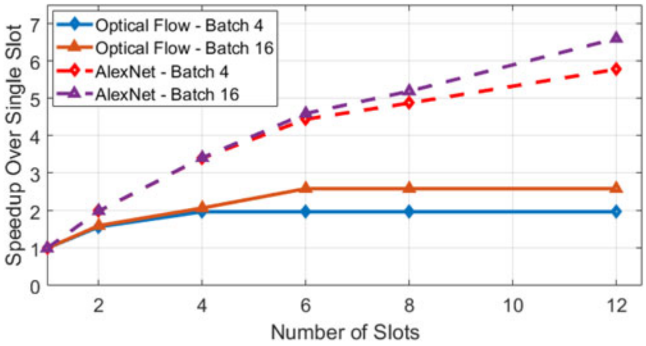


Fig. 20. Effect of number of slots on the performance of Alexnet (AL4) and Optical Flow (OF) benchmarks.

5.7 Impact of Number of Slots on Scheduling Multiple Applications

We have just shown that the best multi-app scheduling scheme is highly dependent on the topology of DAGs in the applications. To further quantify this phenomenon, Fig. 20 shows the performance impact of increasing the number of slots for two different batch sizes for our two applications with the most number of IPs– AlexNet and Optical Flow – both run with pipelining and no parallel batching. AlexNet has 38 IPs with many parallel branches, while Optical Flow has 9 IPs and has no parallel branches. Fig. 20 shows the performance of AlexNet continues to increase as we increase the number of slots. This is due to the fact that AlexNet has many parallel branches, so it can utilize all slots provided and provides a maximum speedup of 6.60X when using twelve slots. On the other hand, Optical Flow sees no increase in performance after four and six slots when using a batch of four and sixteen, respectively. This is because there are no branches in Optical Flow, and the scheduler can reuse the slots and obtain the same performance. Increasing the batch size increases the number of IPs which can run concurrently, however, even with a batch of 16 the speedup when using twelve slots is only 2.60x, despite there being nine IPs in the application. This trend can be used to help determine if dependent or independent scheduling is better for a given set of applications.

5.8 Comparison With Previous Work

We compare our work with an ILP-based approach and heuristic optimized approach [4], [5] – Iterative Scheduling (IS), which limits the number of tasks per itertation to 1 (IS-1) and 5 (IS-5). The results of this comparison can be seen in Table 4. Since we have used a novel architecture in this work, and real-world benchmarks, it is difficult to perform a fair one-to-one comparison with [4], [5] where they use

TABLE 4
Comparative Performance of Our ILP Solution

This Work		[5]			[4]
Nodes	Time(s)	Num Nodes	IS-1	IS-5	PAR/IS-5
7	3.6	10	0.95	34.70	4.730
28	18.06	30	26.02	764	90.30
56	39.8	50	115.6	393	135.36

synthetic benchmarks. Thus, we use our four synthetic test benchmarks as well, and target the same system: a Zedboard with four 1X slots. An ILP-based solution will provide an optimal (or near-optimal for the iterative schedulers in [5] and [4]) solution. Thus, we restrict our comparison to ILP runtimes only, for similar number of nodes in the graphs. Since [5] and [4] do not have pipelining support, we set our batch size to one to disable any pipelining optimization. As we can see, the simplification of our ILP constraints allows us to tackle large problems faster and more efficiently, hence proving the efficacy of our approach. Note that IS-1 is faster than us; however, the approach schedules one task in the queue at a time, which limits the quality of the solution. In contrast, we attempt to schedule up to 25 task nodes first, before partitioning the graph into cuts of 15 tasks.

6 Conclusion

In this work we presented *DML*, an end-to-end DPR scheduling and mapping solution. DML is generic, considers FPGAs of all sizes, provides a scalable and portable architecture that reduces design effort, and includes a novel ILP-based scheduler that provides the lowest latency schedule, performs pipelining and parallelization across batch elements, and is capable of simultaneously scheduling and mapping multiple applications at once.

We demonstrated the efficacy of our solution via an extensive design space exploration via our scheduler, and validated our methodology on edge and cloud scale FPGAs – a Zedboard and a ZCU106. Our evaluation demonstrated our scheduler's ability to pipeline and parallelize the solution with an average speedup of 5X and up to 7.65X on a ZCU106. Finally, we explored the trade-offs between simultaneously mapping multiple applications to a single FPGAs versus partitioning and allocating resources to each application, individually.

Acknowledgments

Edward Richter and Mang Yu have contributed equally to this work.

References

[1] B. Hutchings and M. Wirthlin, "Rapid implementation of a partially reconfigurable video system with PYNQ," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2017, pp. 1–8.

[2] D. Koch *et al.*, "Partial reconfiguration on FPGAs in practice - Tools and applications," in *Proc. IEEE ARCS*, 2012, pp. 1–12.

[3] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware task scheduling for partially reconfigurable FPGAs," in *Proc. Appl. Reconfigurable Comput.*, 2015, pp. 487–498.

[4] A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, and M. D. Santambrogio, "Resource-efficient scheduling for partially-reconfigurable FPGA-based systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 189–197.

[5] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio, "A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2015, pp. 1–6.

[6] R. Cordone *et al.*, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 5, pp. 662–675, May 2009.

[7] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proc. Des. Automat. Test Eur.*, 2008, pp. 519–522.

[8] F. Redaelli *et al.*, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, 2008, pp. 97–102.

[9] A. Agne *et al.*, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan./Feb. 2014.

[10] A. Rodr íguez *et al.*, "FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICO3 framework," *Sensors*, vol. 18, no. 6, 2018, Art. no. 1877.

[11] B. Seyoum *et al.*, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in FPGA soc," in *Proc. Annu. ACM Symp. Appl. Comput.*, 2021, pp. 481–490.

[12] M. Rabozzi *et al.*, "Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 25, no. 1, pp. 151–164, Jan. 2017.

[13] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2018, pp. 269–278.

[14] Xilinx. Zynq-7000 SoC Data Sheet: Overview. Accessed: Jul. 2, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[15] Xilinx. Zync UltraScale MPSoC Data Sheet: Overview. Accessed: May 26, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf

[16] Xilinx, *Partial Reconfiguration Flow on Zynq using Vivado*, [Online]. Available: https://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-partial-reconfiguration-flow-zynq.html

[17] S. DiamondandS. Boyd, "CVXPY: A python-embedded modeling language for convex optimization," *J. Mach. Learn. Res.*, vol. 17, no. 83, pp. 1–5, 2016.

[18] Gurobi optimizer, Gurobi, v8.1. [Online]. Available: https://www.gurobi.com/products/gurobi-optimizer/

Ashutosh Dhar (Member, IEEE) received the MS and PhD degrees from the University of Illinois, Urbana-Champaign. His research interests include computer architecture, reconfigurable, heterogeneous architectures, and application of reconfiguration in conventional architectures.

Edward Richter (Student Member, IEEE) received the BS and MS degrees in electrical and computer engineering from the University of Arizona. He is currently working toward the PhD degree with the Electrical and Computer Engineering Department, University of Illinois, Urbana-Champaign. His research interests include utilizing reconfigurable computing platforms for acceleration and enabling architectural and system-wide research.

Mang Yu received the BS and MS degrees from the University of Illinois, Urbana-Champaign. His research focuses on utilizing machine learning methods to improve accelerator designs for reconfigurable computing platforms.

Wei Zuo received the MS degree from the Electronics and Communication Engineering Department, University of Illinois, Urbana-Champaign, where she is currently working toward the PhD degree with Electrical and Computer Engineering Department. Her research interests include hardware-software co-design, developing automated frameworks for accurate system-level modeling, and efficient hardware–software partitioning for applications mapped to SoC platforms.

Xiaohao Wang received the BS and MS degrees from the University of Illinois, Urbana-Champaign. His research focuses on hardware systems.

Nam Sung Kim (Fellow, IEEE) is currently a professor with the University of Illinois, Urbana-Champaign. He has authored or coauthored more than 200 refereed articles to highly-selective conferences and journals in the field of digital circuit, processor architecture, and computer-aided design. The top three most frequently cited papers have more than 4000 citations and the total number of citations of all his papers exceeds 11,000. He was the recipient of the IEEE International Symposium on Microarchitecture (MICRO) Best Paper Award in 2003, NSF CAREER Award in 2010, and ACM/IEEE Most Influential *International Symposium on Computer Architecture* (ISCA) Paper Award in 2017. He is a hall of fame member of IEEE International Symposium on High-Performance Computer Architecture (HPCA), MICRO, and ISCA. He is a fellow of ACM.

Deming Chen (Fellow, IEEE) received the BS degree in computer science from the University of Pittsburgh in 1995, and the MS and PhD degrees in computer science from the University of California, Los Angeles, in 2001 and 2005, respectively. He is currently the Abel Bliss professor of engineering with the Electronics and Communication Engineering Department, University of Illinois, Urbana-Champaign (UIUC). His research interests include system-level and high-level synthesis, machine learning, computational genomics, reconfigurable computing, and hardware security. He was the recipient of the UIUC's Arnold O. Beckman Research Award, NSF CAREER Award, nine best paper awards, ACM SIGDA Outstanding New Faculty Award, and IBM Faculty Award. He is an ACM distinguished speaker and the editor-in-chief of *ACM Transactions on Reconfigurable Technology and Systems*.