CLARE$^{\mathrm{RT}}$: Deterministic Cycle-Level Accelerator on REconfigurable platforms in DNN-Enabled Real-Time Safety-Critical Systems

Abstract—**Deep neural network (DNN) models are increasingly deployed in real-time, safety-critical systems such as autonomous vehicles, driving the need for specialized AI accelerators. However, most existing accelerators support only non-preemptive execution or limited preemptive scheduling at the coarse granularity of DNN layers. This restriction leads to frequent priority inversion due to the scarcity of preemption points, resulting in unpredictable execution behavior and, ultimately, system failure.**

To address these limitations and improve the real-time performance of AI accelerators, we propose CLARE$^{\mathrm{RT}}$, a novel accelerator architecture that supports fine-grained, intra-layer flexible preemptive scheduling with cycle-level determinism. CLARE$^{\mathrm{RT}}$ incorporates an on-chip Earliest Deadline First (EDF) scheduler to reduce both scheduling latency and variance, along with a customized dataflow design that enables intra-layer preemption points (PPs) while minimizing the overhead associated with preemption. Leveraging the limited preemptive task model [1]–[3], we perform a comprehensive predictability analysis of CLARE$^{\mathrm{RT}}$, enabling formal schedulability analysis and optimized placement of preemption points within the constraints of limited preemptive scheduling. We implement CLARE$^{\mathrm{RT}}$ on the AMD ACAP VCK190 reconfigurable platform. Experimental results show that CLARE$^{\mathrm{RT}}$ outperforms state-of-the-art designs using non-preemptive and layerwise-preemptive dataflows, with less than 5% overhead in worst-case execution time (WCET) and only 6% additional resource utilization.

I. Introduction

Deep Neural Network (DNN) inference has become a cornerstone of modern intelligent systems, powering applications such as autonomous vehicles [4], drones [5], and robotics [6]. Many of these applications perform safety-critical tasks that require real-time inference, and AI accelerators have emerged as a key computing platform for efficiently executing DNN workloads.

While dedicating an entire accelerator to serve a single DNN application can meet stringent low-latency requirements, this approach does not support concurrent execution or flexible context switching, capabilities that are standard on CPU cores. To improve resource utilization and system responsiveness, it is increasingly common to share accelerators among multiple applications with varying timing constraints. For instance, autonomous vehicles simultaneously rely on DNNs to detect other vehicles, pedestrians, traffic signs, and signals in all directions - each a real-time safety critical task.

Scheduling DNN inference on shared accelerators introduces two major challenges. First, real-time tasks may differ in their runtime importance and can have dynamically changing priorities. Second, to meet real-time guarantees, all critical tasks must be served concurrently on the accelerator. State-of-the-art accelerators address concurrency by offering multiple execution streams. However, due to the non-preemptive nature of the accelerator, all tasks are scheduled as a First-In-First-

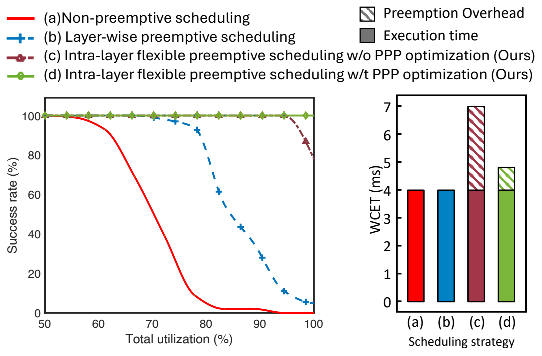


Fig. 1: Non-Preemptive vs. Preemptive Scheduling.

Out (FIFO) queue [7], [8]. In this solution, a high-priority task in the queue can be blocked by others and is prevented from being executed. This phenomenon, known as priority inversion, often results in unpredictable and unacceptable performance in time-sensitive scenarios and leads to deadline missing, leading to the failure of the whole system. Figure 1 shows a group of task sets with two tasks under different randomly generated utilization distributions. The success rate under different total utilization and worst-case execution times (WCET) of different designs are reported. As shown in Figure 1 strategy a, such non-preemptive scheduling easily fails when the total utilization is high.

Inspired by the way CPUs provide real-time guarantees through preemptive scheduling [1], [2], a natural extension is to incorporate preemption into accelerator scheduling. However, existing accelerators typically lack support for flexible and fine-grained preemption. Some accelerators, such as GPUs, support only layer-wise preemption, leveraging the limited preemption opportunities that occur between layers of DNN models. Yet, as illustrated in Figure 1 (denoted by strategy b), this approach offers only a limited number of inter-layer preemption points. Consequently, as system utilization increases - particularly in the 80–90% range - the success rate of task scheduling significantly declines.

To address these challenges, we present CLARE$^{\mathrm{RT}}$, a novel accelerator architecture that enables fine-grained, intra-layer preemptive scheduling. CLARE$^{\mathrm{RT}}$ incorporates customized memory interfaces, on-chip buffers, a flexible dataflow controller, and a kernel management module to support efficient preemption and resumption within DNN layers. With intralayer preemption points (PP), CLARE$^{\mathrm{RT}}$ achieves higher scheduling success rates than baseline non-preemptive and layerwise-preemptive dataflows (strategies c and d in Figure 1). However, scheduling and preemption incur non-trivial overhead (strategy c), especially for short inference tasks.

To reduce preemption overhead, CLARE$^{\mathrm{RT}}$ dynamically chooses between two strategies: (1) discarding and recomputing intermediate data, avoiding DRAM communication but adding compute cost; or (2) persisting data to DRAM and reloading it, avoiding recomputation but incurring memory access. A flexible dataflow engine selects the optimal strategy at each preemption point, balancing performance and efficiency.

Note that unlike CPU-based schedulers that suffer from cache-induced variability, CLARE^{RT} implements a cycle-accurate, finite state machine (FSM)-based earliest-deadline-first (EDF) scheduler on programmable logic (PL), ensuring bounded worst-case latency per operation. CLARE^{RT}'s deterministic cycle-level behavior enables analytical performance modeling. We adapt existing limited preemptive scheduling theory [1], [3] to support PP placement and schedulability analysis tailored to CLARE^{RT}. A heuristic algorithm is applied for preemption points placement (PPP) optimization by removing redundant points to minimize overhead [2]. Evaluation results demonstrate that CLARE^{RT} achieves stable and bounded end-to-end latency under real-time constraints (i.e., strategy d in Fig. 1). In summary, our contributions are as follows:

- **Finite state machine-based scheduler:** We implement a EDF scheduler on programmable logic using a FSM design, ensuring low-latency and cycle-level deterministic scheduling with bounded overhead and minimal variance.
- **Intra-layer preemptive accelerator design:** We design the CLARE^{RT} accelerator with customized architecture and dataflow to enable fine-grained preemption within a DNN layer.
- **Flexible dataflow for preemption:** To optimize the cost of preemption operation, we introduce a flexible dataflow mechanism that dynamically selects between persisting or recomputing intermediate data, leveraging the strengths of both strategies.
- **Performance modeling, predictability analysis, and PPP optimization:** We develop a performance model for CLARE^{RT} and adapt limited preemptive scheduling theory to enable formal schedulability analysis. Additionally, we propose a heuristic for joint PP placement and dataflow strategy selection to reduce preemption overhead.

II. Related Works

Preemptive scheduling on accelerators has been explored in a number of prior works, many of which target GPU platforms. These works enable preemptive scheduling either by partitioning the workloads into subtasks [9], [10] or by implementing the context-switch mechanisms [11]. However, the CPU and GPU accelerators apply cache-based architecture, leading to lower determinism in execution times. On the other hand, in the FPGA-based accelerators, memory access is explicitly controlled, resulting in a deterministic latency.

Several frameworks have been proposed for the FPGA platforms with real-time system-related techniques like preemption or deadline-aware scheduling, which usually target multi-tenancy systems with quality-of-service (QoS) requirements. [12]–[15] implement the scheduling algorithm but do not support preemption, thus their executions of task sets are non-preemptive. CD-MSA [16] proposes a deadline-aware scheduling mechanism that uses the off-chip memory access phase between layers, i.e., after data is written by one layer and read by the next, as an opportunity for preemption. These works only support layer-wise preemption and lack support for fine-grained intra-layer preemption. Moreover, these frameworks are not designed for real-time, safety-critical systems and thus lack mechanisms to guarantee schedulability.

Some other works discuss schedulability analysis. [17] implements a driver assistance system based on Vitis AI framework [18]. However, the execution of workloads is non-preemptive. ART [19] implements an EDF scheduling layer-wise limited preemption mechanism. MESC [20] implements a context-switch mechanism based on the gemmini accelerator [8]. MESC supports preemption in the granularity of gemmini instructions, which is also layer-wise preemption. These works do not customize the dataflow and are unable to handle intermediate data during execution, making them lack support for intra-layer preemption. In contrast, CLARE^{RT} introduces a dataflow-aware methodology that enables fine-grained control over execution, allowing flexible and efficient intra-layer preemption.

CLARE^{RT} and the works mentioned above all target the DNN workloads by implementing a unified accelerator on the platforms. Different workloads are handled by supplying distinct control signals to the accelerator. Alternatively, partial reconfiguration can be used, in which each workload is mapped to a dedicated bitstream and the FPGA is reprogrammed accordingly whenever a new task is invoked. [21]–[23] discuss the preemption overheads using partial reconfiguration. FRED [24], [25] proposes a framework for real-time safety-critical systems on FPGA accelerators using partial reconfiguration, and DART [26] optimizes the dynamic partial reconfiguration partitioning and floorplanning problem. The cost of partial reconfiguration in these approaches is substantial, and they are unable to support task preemption within the execution of a single bitstream.

III. CLARE^{RT} Hardware architecture

We propose the CLARE^{RT} hardware architecture to achieve cycle-level deterministic control and accelerate operations in limited preemptive systems. Such deterministic features are enabled by the specifically designed heap-based scheduler and the control flow, as well as the intra-layer preemptive flexible dataflow accelerator design. These techniques are customized especially for real-time, safety-critical systems. In this section, we explain the overall CLARE^{RT} architecture and the microarchitectures of each module, including the heap-based scheduler, kernel management module, flexible dataflow controller, and accelerator design. We demonstrate the execution procedure of the system based on the intra-layer preemptive flexible dataflow. The performance model and schedulability analysis of CLARE^{RT} are explained in section IV.

A. Programming Model of Baseline Accelerator on Reconfigurable Platforms

We first discuss the programming model of a baseline DNN accelerator, where different layers in the application will

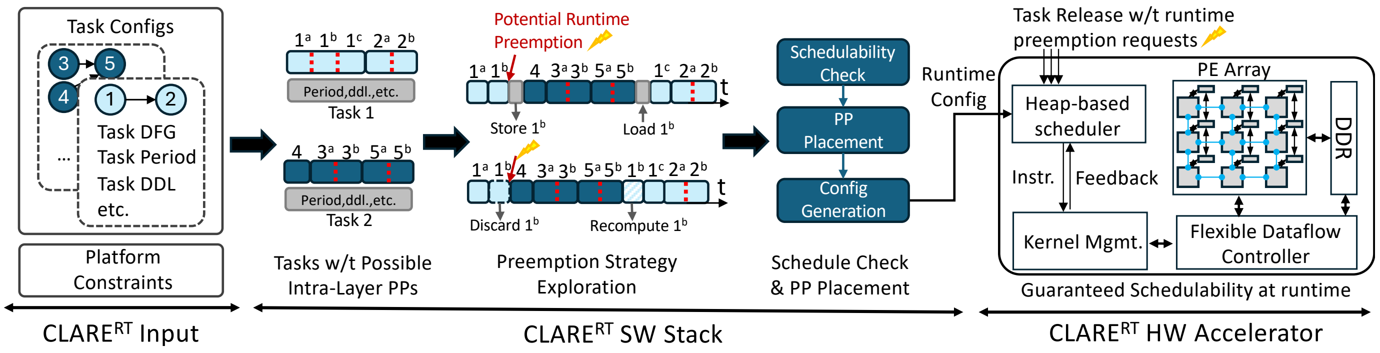


Fig. 2: CLARE^{RT} system architecture.

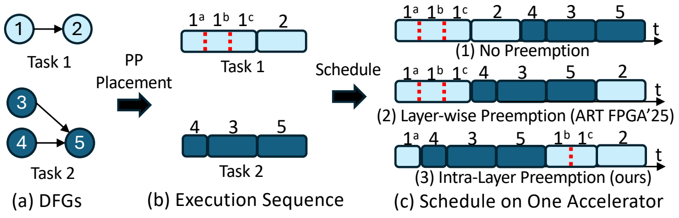

Task 1
Task 2
PP Placement
Schedule
(1) No Preemption
(2) Layer-wise Preemption (ART FPGA'25)
(3) Intra-Layer Preemption (ours)
(a) DFGs
(b) Execution Sequence
(c) Schedule on One Accelerator

Fig. 3: We illustrate three scheduling strategies for executing two DNN tasks on a single accelerator, assuming Task 2 is released immediately after Task 1 starts and has higher priority. (a) shows the dataflow graphs (DFGs) of Task 1 and Task 2; (b) depicts execution timelines with preemption points at the first layer of Task 1; (c) compares three strategies: (1) Non-preemptive scheduling, where Task 2 is blocked until Task 1 completes, resulting in the highest end-to-end latency for Task 2; (2) Layer-wise preemptive scheduling (ART FPGA'25 [19]), which allows preemption only between layers and moderately reduces Task 2's latency; and (3) Intra-layer preemptive scheduling (our approach), which enables fine-grained preemption within layers, allowing better task interleaving and yielding the shortest latency for Task 2 across all strategies.

be executed sequentially by reusing a single accelerator. An illustration of the whole workflow is shown in Figure 3. In the beginning, the DNN models to be executed will be entered in the form of data flow graphs (DFG) (Figure 3 (a)). Since there is only one accelerator processing one layer at a time, an execution sequence (Figure 3 (b)) of each model will be translated from the DFG with the dependency reserved. To improve data locality, tiling is usually applied to break a large computation into smaller chunks that fit into faster on-chip memory (e.g. $1^a, 1^b, 1^c$ in Figure 3 (b)). Due to varying layer shapes, each layer is partitioned into a different number of tiles. Consequently, the tasks will be scheduled onto the accelerator for execution. This is achieved by sending the control signals to the accelerator, which usually contains information about DDR addresses and the loop boundary. The accelerators will process the tiles iteratively following the schedule, and the granularity of execution is determined by the architecture and data flow of the accelerator. Without customized control over intermediate data, the baseline accelerator treats an entire DNN layer as the minimum granularity, processing all tiles sequentially without support for fine-grained, tile-level control. To bridge the gap between hardware acceleration and intra-layer preemption support, we introduce a novel accelerator architecture with a small overhead in Section III-B.

B. *CLARE^{RT} System Overview*

The overview of CLARE^{RT} is shown in Figure 2, which includes the software stack and hardware accelerator. Beginning with a given task set with several tasks together with the platform constraints like on-chip resources, CLARE^{RT} will first generate the fine-grained execution pattern based on the problem specifications of each layer in each task. Coupled with specialized data flow accelerator design, CLARE^{RT} is intra-layer preemptive, i.e., a high-priority task can preempt the ongoing one even if the accelerator is executing within a layer. For each preemption point, since CLARE^{RT} applies dynamic scheduling where the preemption happens at runtime, thus two strategies of context switch exist: (1) **persist** the on-chip intermediate data to DDR, and load it back when resuming, and (2) drop the intermediate data, and **recompute** it when resuming.

Based on the accelerator design, and the position of the preemption point, these two strategies represent tradeoff in preemption overheads. CLARE^{RT} use its performance model to give cycle-level accurate estimation on the Worst Case Execution Times (WCETs), PP positions, and preemption overheads of both strategies. Such accuracy can be achieved since the performance model is directly derived from the design, different from existing works like [17] where the model is simulation- or profiling-based. With all the information from the input task set and from the performance model, we conduct schedulability analysis together with the PP placement algorithm to get the final schedule with schedulability checked and redundant PP removed. The system can only preempt a task at a placed PP, and the region of workload between two consecutive PPs can not be preempted, forming the non-preemptive regions (NPRs). We make adjustments to this algorithm so that it can be applied to realistic schedulers and accelerators, where hardware overhead is ineliminable. The decision of the dataflow strategy (persist vs. recompute) of each PP is also determined together with the PP placement. The resulting schedule is also cycle-level accurate and can guarantee the schedulability if the test is passed. The schedule will then be used to build up the CLARE^{RT} hardware accelerator.

On the hardware side, to control the whole system, CLARE^{RT} does not rely on a CPU nor an operating system

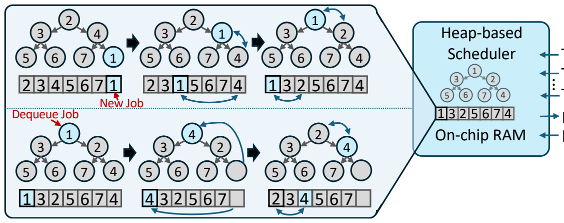


Fig. 4: Heap-based scheduler implementated in hardware.

```
//CLARE heap operation (Vitis HLS)
void insert_job(job_t* heap, job_t new_job, int &tail){
   #pragma HLS inline off
   //insert the new job at the tail of the heap
   heap[tail] = new_job;
   tail++;
   //bottom-up maintenance: swap from tail to root
   for(int c=tail; c>0; c=(c-1)/2){
   #pragma HLS pipeline II=2
      //compare and swap the current node with its parent
      if(heap[c].ddl<heap[(c-1)/2].ddl){
         swap(heap, heap[c],heap[(c-1)/2];}
      else{
         break;
      }}
void dequeue_job(job_t* heap,int tail){
   #pragma HLS inline off
   //dequeue the root (job has finished)
   heap[0] = heap[tail];
   tail--;
   //top-down maintenance: swap from root to tail
   for(int p=0;2*p+1<=tail;){
      #pragma HLS pipeline II=3
      int smallest_node = p;
      //compare and swap the current node with its children
      if(heap[2*p+1].ddl<heap[p].ddl
         && heap[2*p+1].ddl<heap[2*p+2].ddl){
         smallest_node = 2*p+1;}
      else if(heap[2*p+2].ddl<heap[p].ddl
         && heap[2*p+2].ddl<heap[2*p+1].ddl){
         smallest_node = 2*p+2;}
      if(smallest_node!=p){
         swap(heap, smallest_node, p);
         p=smallest_node;}
      else{
         break;
      }}
```

Fig. 5: Heap operation implementation in HLS.

to run the scheduling, instead, an in-house scheduler is implemented on the FPGA side as a finite-state machine(a). We design a heap-based scheduler architecture on the FPGA to manage released jobs. This design enables the scheduler to efficiently track job deadlines, supporting Earliest Deadline First (EDF) scheduling. The heap structure of the scheduler ensures that every operation of the scheduler is predictable in cycles and the worst-case overhead is deterministic. With EDF scheduling, each time the scheduler picks a non-preemptive region (NPR) to execute, then the kernel management mechanism translates the `(task, region)` tuple to the metadata that instructs the accelerator to execute. The metadata is tailored for different task sets, contains the problem shapes (e.g., MKN in a matrix multiplication), addresses, the PP placement information, and the strategy for preemption. This gives CLARE$^{\text{RT}}$ flexibility to apply different strategies in different PPs. The synchronization pattern between the scheduler and the kernel management module is also customized to enable scheduling and acceleration runs simultaneously to reduce overhead and avoid deadlocks that can happen.

For all of the programmable logic (PL) implementations, we use high-level synthesis (HLS) for better productivity. We propose a clear and reusable HLS coding style and provide detailed explanations about how to generate deterministic high-performance designs that allow other designers to adopt and apply this approach as well.

C. Heap-based Scheduler

Given N tasks, the functionality of an EDF scheduler in the limited preemptive scenario includes (1) recording the jobs released by each task (2) picking the job with the smallest deadline and issuing it to the accelerator, when a non-preemptive region finishes(i.e., reaching a preemption point) (3) updating the job record accordingly to track the progress of each job and cleaning the job when all its regions are finished.

In CLARE$^{\text{RT}}$, the status of the released jobs is recorded in an array. A tail register and two workers are added to maintain the array as a minimum heap so that the first element of the array will always have the smallest deadline after maintenance. This also helps to reduce the hardware operation execution time needed to find the most urgent job. For a task set of N, the size of the heap can be fixed at N since under the limited preemption scenario, two jobs of one single task exist simultaneously means at least one job violates the deadline. The task release is implemented using FIFOs and we assign an independent channel for each task. Additionally, the feedback and instruction channels are implemented to interact with the accelerator, and a main FSM is used to control all the functionalities.

1) *Heap Operations:*

A min-heap as an array requires the following heap property: all the parent nodes (indexed as i) to be smaller than both its children nodes(indexed as 2*i+1 and 2*i+2). Two workers are implemented using HLS to maintain the heap in a top-down or bottom-up manner, as shown in Figures 4 and 5. When a new job is released, it is first inserted into the tail of the heap, then the worker iterative from tail to root, swapping the current node with its parent when it has a smaller deadline. After scheduling the last region of one job (the job must be at the root since it has the smallest deadline), the worker moves the job at the tail of the heap to the root, then iterates from root to tail, swapping the node with its smallest children for heap property.

Several HLS techniques are applied to optimize the heap operations. The `inline off` pragma directs the HLS compiler to compile these two functions as individual modules that can be invoked by multiple functions. The `pipeline` pragma directs the loop body to be implemented in a pipeline manner. With a successful pipelining, the hardware module will process loops in every II (initial interval) cycles, while for each loop it will still take *depth* cycles from start to end. Both of these pipeline parameters can be obtained once the HLS compile is finished, meaning that the heap operations are completely deterministic.

2) *Control Logic Implementation:*

Figure 6 demonstrates the logic implemented in HLS to handle different events. Several functionalities are integrated into this control logic.

Main Loop Structure: The main body of the control logic is an unbounded loop. We also implement logic to safely terminate the whole system for debugging and testing purposes. In this loop, all logic is within different branches of a

```
//CLARE scheduler control logic
job_t job_array[N];
#pragma HLS ARRAY_PARTITION variable=job_array complete
int cur_task, cur_region;
bool issue_flag = 1;
for(int i=0;;i++){//main loop
      //Event: get feedback
      if(!feedback.empty()){
         feedback_buf = feedback.read();
         issue_flag = true;
         instr_idx++;
      }
      //Event: task releases
      else if(!task0_release.empty()){
         release_buf = task0_release.read();
         insert_job(job_array,N,tail,
         {0,0,release_buf.ddl});
         tail++;
      }
      /*task 1,2,...,n release: omitted*/
      //Event: issue instr
      else if(issue_flag && tail>-1){
         instr.write({job_array[0].task,job_array[0].region,
         /*preempt=*/job_array[0].task!=cur_task &&
              cur_region<region_num_array[cur_task]},
         /*resume=*/job_array[0].task!=cur_task &&
              cur_region!=0
         /*last_t=*/cur_task,/*last_r=*/cur_region);
         cur_task = job_array[0].task;
         cur_region = job_array[0].region;
         job_array[0].region++;
         //all regions finished, dequeue job
         if(job_array[0].region >
              region_num_array[job_array[0].task]){
               dequeue_job(job_array, N,tail);
               tail--;
            }
         issue_flag = false;
      }
```

Fig. 6: Scheduler control logic implementation in HLS.

single if statement. For the event of feedback from accelerators and task releases, which are related to the input FIFO, we check one FIFO in one if branch, and only perform operation when the FIFO is not empty. This prevents the HLS compiler from grouping the checking of different FIFOs into one combinational logic, which can lead to wrong implementation of FSM and deadlock at runtime. That is, the FSM tries to read the feedback and issue the instruction in the same state and checks both FIFOs simultaneously, as a result, it has to get feedback before issuing an instruction. Besides, the main loop can proceed when there is no feedback or the issue flag is false. This means the scheduler can handle the job release when the accelerator is still running, reducing the critical path between receiving feedback to issuing the next instruction.

Job Release Handling: When a job is released, an element is sent to the FIFO specifying the deadline for this job. The scheduler will add this job to the tail of the heap, and perform the bottom-up maintenance. When multiple jobs are released in the same cycle, the scheduler will process them sequentially in different main loop iterations. This ensures that before and after each task release is processed, the heap property is kept.

Issue Instructions: An instruction will be issued and sent to the FIFO if and only if (1) the `issue_flag` is set to 1 and (2) the array is not empty. This means the scheduler will only send a new instruction after getting feedback, which turns on the `issue_flag` during runtime. The branch of receiving feedback and issuing instructions takes the highest and lowest priority in the loop, as a result, the scheduler will check all job release channels before issuing the next instruction. This prevents the case that a job with the smallest deadline is released a few cycles before the feedback arrives, but is not added to the heap thus not executed due to the time cost in scheduling. The `issue_flag` is set to true during the initialization process before the main loop, meaning that the scheduler will issue the first instruction when the first jobs are released to start up the whole system.

When issuing an instruction, the scheduler simply picks the job at the top of the heap, which always has the smallest deadline. The instruction has several attributes: the `task` and non-preemptive `region` of the most urgent job, and two bool variables `preempt` and `resume`. These two variables determine if this new region presents as a preemption of the ongoing job or a resume of a job that is executed halfway. Their values are obtained by comparing the recorded ongoing job with the job on top of the heap. The task and region that is issued in the last instruction are also recorded and sent via `last_t` and `last_r`.

After the instruction is sent to FIFO, the corresponding region can be regarded as guaranteed to be finished when receiving the next feedback. Thus it is safe to update the heap immediately after issuing the instruction. For the jobs where the issued region is not the last one, only the top node needs to be changed to the next non-preemptive region. If the issued region is the last one, the top node will be dequeued out, and a new job with the smallest deadline will be swapped to the root using heap operations.

3) Latency Analysis: For the two heap operations, the $depth$ of top-down and bottom-up maintenance are 4 and 5, respectively, and the II is set as 3 and 2. For a task set of N, the heap size is also provisioned as N, constructing a $\lceil \log(N) \rceil$-level heap. The loop boundary of the heap operations is also $\lceil \log(N) \rceil$. As a result, the worst case of the heap operation is bound to $3\lceil \log(N) \rceil + 1$ and $2\lceil \log(N) \rceil + 3$ cycles for top-down and bottom-up maintenance operations, where the maintenance goes across all levels of the heap.

The latency of each branch in the main loop is also deterministic. In the compiled FSM, checking the if statement takes one state (denoted as state A), and each branch body takes one state or several states in sequence. It is the design that determines the latency of each state in the FSM, thus the latency of the whole scheduler is also bounded. Following the setup in III-C1: (1) the feedback branch takes 2 cycles at most; (2) each task release branch takes $2\lceil \log(N) \rceil + 5$ cycles at most, assuming the heap operation reaches the worst case; (3) the issue instruction branch takes $3\lceil \log(N) \rceil + 4$ cycles at most, assuming the last region of a job is issued, and heap operation reaches the worst.

D. Accelerator Microarchitecture

Figure 7(c) shows the dataflow structure of the accelerator, which is managed by a flexible dataflow controller. The tiling strategy is used for processing different layer sizes. For a Matrix Multiplication of arbitrary size, it can be partitioned to a set of tiles with uniform size TM$\times$TK$\times$TN. We apply output stationary dataflow for the accelerator, that the subtiles along the reduced dimension (K) will be accumulated in the on-chip buffer completely before being stored in the off-chip memory. Pipelining is also applied, allowing the accelerator to load/-

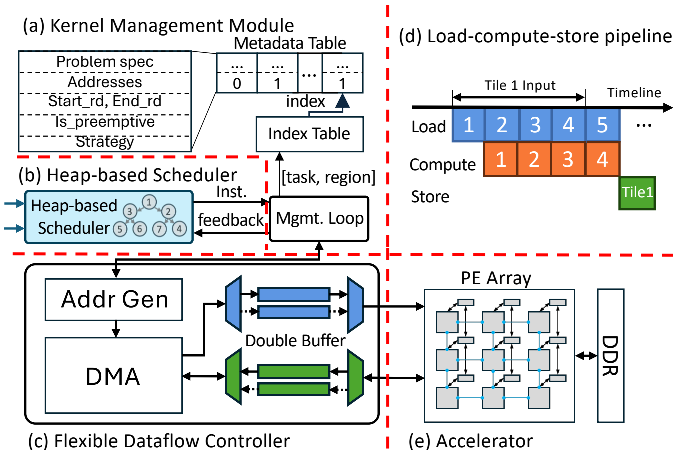


Fig. 7: CLARE^{RT} hardware accelerator system.

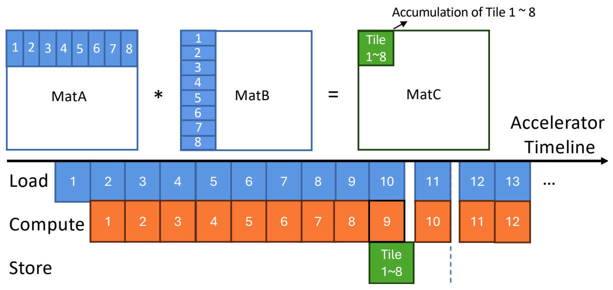


Fig. 8: Execution pattern without preemption happening.

compute/store different tiles and overlap the communication and computation latency. As a result, the execution of a layer can be partitioned into a sequence of iterations, as shown in Figure 7(d). To avoid of the dependencies, each iteration will load, compute, and store data of different tiles, and the latency is bounded by the slowest operation. For a matrix multiply of shape $M \times K \times N$, there will be $M/TM \times K/TK \times N/TN$ tiles and $M/TM \times K/TK \times N/TN + 2$ iterations where extra 2 is for entering and draining pipeline.

Latency for load and store from DDR: Different from the designs on CPUs and GPUs, the address generator and AXI-interface of CLARE^{RT} is implemented in FPGA, where designers have full control of issuing DMA instructions. As a result, the latency of loading and storing the input and output buffer is bounded. We discuss the performance model for DDR access in Section IV-C.

Latency for computation: The cycle used in computation varies due to different PE array designs. However, for a certain design, the cycle number is fixed. CLARE^{RT} uses the performance model of the PE array from the original design.

E. Kernel Management Module and Metadata

As shown in Figure 7(a), the kernel management module receives the instruction from the scheduler specifying a non-preemptive region and then controls the accelerator to execute this region. All information for all non-preemptive regions within the given task is stored on a static on-chip metadata table, and a 2-D indexing table is implemented to index the start of each non-preemptive region. One non-preemptive region can contain different layers with different shapes and addresses, they are stored in consecutive entries in the metadata table, using a `is_preemptive` bit to suggest the current non-preemptive region after finishing this segment. When the kernel management module receives an instruction, it will locate the first entry of the non-preemptive region using the index table, then sequentially execute the entries in the metadata table until reaching the end (i.e., `is_preemptive==1`). After finishing the last entry, a 1-bit feedback will be sent to the scheduler. The accelerator will wait for the scheduler to send the next instruction, and preemption happens if the new instruction contains a region of a different task, while the ongoing task has not finished yet.

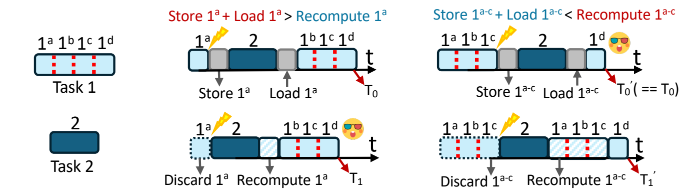
The metadata should also be able to handle a "partial" layer so that we can place the preemption point within a layer. In CLARE^{RT} we implement this functionality by storing the start and stop iterations of this non-preemptive region in the metadata table. This is based on the observation that, once the shape of a layer is given, the number of its partitioned tile is determined. For each iteration, which tiles it loads/computes/-stores for, the choice of using which ping-pong buffer, and the address of loading/storing the tile are determined and can be inferred from the iteration index.

Latency Analysis: After receiving instruction in the 1st cycle, it takes only one cycle to find the corresponding metadata, and the kernel is launched at the 3rd cycle. Then each time new metadata is needed when running a non-preemptive region, it takes one additional cycle for metadata. After the region finishes, the feedback will be sent in the second cycle.

F. Intra-layer Preemptive Flexible Dataflow

By designing an architecture specifically tailored for real-time, safety-critical scenarios, CLARE^{RT} supports intra-layer preemption, with preemption points positioned between the execution of two consecutive tiles. Besides, for each preemption point, CLARE^{RT} provides two design choices of either persisting the intermediate data to DDR and loading back upon resuming, or discarding the intermediate data and recomputing upon resuming. These two strategies represent a tradeoff in the cost of preemption operations. Based on our proposed cycle-accurate performance model, CLARE^{RT} automatically adopts the best strategy to reduce the costs of the whole system.

1) Execution Procedure without Preemption: We demonstrate how the customized flexible dataflow works in preemption and resuming in the system through case studies of synthetic workloads. Figure 8 shows an execution scheduling with no preemption. For the matrix matmul workload, we apply tiling and adopt output stationary, where the tiles $1 \sim 8$ from MatA and MatB are loaded in each iteration, and their partial products are accumulated into the same output tile. The accelerator will store one output tile every 8 iterations. We partition the workload such that the first non-preemptive region of task 1 contains regions 1 to 11. At first, both the scheduler and accelerator are idle, and the job of task 1 is released. This job will be added to the heap-based scheduler, which then issues its first region as the first instruction. This scheduling process only takes a few cycles, then the accelerator will be launched to execute the first region for 11 iterations. At this time constant, the on-chip intermediate data contains: (1) the input data of tile 11 in the input buffer, and (2) the partial



1a 1b 1c 1d
Task 1
2
Task 2
Store 1a + Load 1a > Recompute 1a
1a 2 1b 1c 1d t
Store 1a Load 1a T0
1a 2 1b 1c 1d t
Discard 1a Recompute 1a T1
Store 1a-c + Load 1a-c < Recompute 1a-c
1a 1b 1c 2 1d t
Store 1a-c Load 1a-c T0′(== T0)
1a 1b 1c 2 1a 1b 1c 1d t
Discard 1a-c Recompute 1a-c T1′

(a) Tasks w/t PPs (b) Preemption Case 1: CLARE^{RT} chooses recompute (c) Preemption Case 2: CLARE^{RT} chooses persist, i.e., store+load

Fig. 9: Illustration for the trade-off between recompute and persist strategies in intra-layer preemption. (a) Task 1 is inserted with PPs, allowing intra-layer preemption, while Task 2 is a separate task with higher priority; (b) In preemption case 1, CLARE^{RT} selects the recompute strategy because the execution time of storing and loading intermediate data (1^a) exceeds that of recomputing the intermediate data; (c) In preemption case 2, where CLARE^{RT} selects the persist strategy because the execution time of storing and loading intermediate data (1^{a-c}) is smaller than that of recomputing the intermediate data. CLARE^{RT} flexible dataflow controller chooses either persist or recompute strategy to optimize the cost in terms of the execution time of the preemption operation.

results accumulated from tiles 9 and 10, which partial products contribute to a different output tile than tile $1 \sim 8$. Once the accelerator finishes this partial execution, it sends feedback, which notifies the scheduler to issue the next instruction. Suppose in the second instruction, the `preempt` and `resume` flags are set to 0, suggesting that the second region comes directly after its predecessor, so the accelerator can start from iteration 12.

2) Dataflow Using Recompute Strategy: Figure 9 (b) demonstrates the execution procedure when the recompute strategy is chosen. The Tasks are depicted in Figure 9 (a), where Task 1 has four PPs while Task 2 is a separate task with higher priority. During the execution of the first tile (1^a) of Task 1, the scheduler receives a new job from Task 2. Because Task 2 has a higher priority, once the tile 1^a is finished, the scheduler will preempt Task 1 and issue instructions to execute Task 2. At this time, the on-chip storage will hold the input for tile 1^b and the partial results from tile 1^a. Both of them will be discarded in the recompute strategy. Once Task 2 is finished, Task 1 will be resumed.

Since the intermediate data is not stored, it introduces a cost for recomputing. The overhead mainly comes from the recompute and preemption to reset the output buffer, where the latter has a fixed number of cycles to reset.

In contrast, the persist strategy needs to store the partial results of tile 1^a and then load it back, which takes more time than recomputing. Therefore, CLARE^{RT} will apply the recompute strategy in other similar cases.

3) Dataflow Using Persist Strategy: Figure 9 (c) demonstrates a case where the persist strategy is preferred. As mentioned in Figure 8, the data should be stored only after certain iterations for better efficiency. Therefore, the recompute overhead increases and peaks at the last iteration before storing. In contrast, the persistent overhead (i.e., storing and loading) will not vary as the partial results will be accumulated. These different trends in overhead can alter the preferred preemption strategy. In Figure 9 (c), the preemption signal arrives during the execution of tile 1^c, and the output buffer will be fed by the partial results for tiles $1^a - 1^c$ when the preemption is responded to.

Compared with the recomputing strategy, in which the tile $1^a - 1^c$ must be computed again, the extra time from storing and loading the intermediate results is smaller. As a result, the persist strategy produces better performance.

4) Flexible Dataflow: The two strategies of recompute and persist present a tradeoff that mainly depends on the off-chip bandwidth of the platform. Suppose TM and TN are much larger than TK, which is common when the spatial parallelism is explored along the M and N dimensions in an output-stationary accelerator. The amount of memory transfer for storing and loading the output will be much larger than the inputs, so recomputing tiles could be faster than persisting their partial results. The position of the preemption is another concern: if the preemption happens when the output buffer is accumulated by several tiles, the recomputing overhead prevails, making the persist strategy more efficient.

For each intra-layer preemption point enabled by the CLARE^{RT}, the choice of strategy is determined before runtime, and the schedulability will be proven in Section IV. A metadata table shown in Figure 7 (a) is used for guiding the hardware accelerator to select a dataflow strategy when preemption happens. Specifically, when receiving a new job, the scheduler will generate an instruction including all necessary information for executing a layer to the kernel management module, which is responsible for checking the metadata table accordingly to obtain the best dataflow strategy.

IV. CLARE^{RT} Performance Model and Schedulability Analysis

According to our system design, it is essential to determine task preemption points through formal schedulability analysis. To this end, we first formalize the real-time system implemented by CLARE^{RT} using a limited preemptive task model [1]–[3]. We begin by defining the key parameters that characterize the system's task structure. Next, we present a cycle-accurate performance model that captures the underlying accelerator design and incorporates worst-case execution time (WCET) analysis. Finally, we demonstrate how this performance model is integrated into the schedulability analysis algorithm to guarantee that all task deadlines are met prior to runtime.

A. Task Model

Following the setup in [2], we define the task model under a limited-preemptive scheduling scenario. The system executes a task set τ, consisting of n periodic tasks. Each task is represented as $\tau_i = (e_i, p_i, d_i)$, where $1 \leq i \leq n$. Here, e_i denotes the worst-case execution time (WCET), p_i is the task period - the minimum time between successive releases of τ_i - and d_i is the relative deadline of each job. We assume an implicit-deadline task model, where the deadline equals the period, i.e., $d_i = p_i$, ensuring that each job must complete before the next one is released.

Each task τ_i consists of multiple non-preemptive regions (NPRs), denoted as $\delta_{i,j} = (b_{i,j}, \xi_{i,j})$. Here, $b_{i,j}$ represents

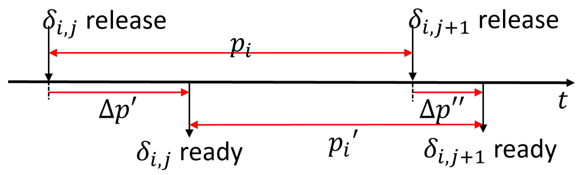


Fig. 10: Timeline of two consecutive jobs.

the execution time of the j-th non-preemptive region, and $\xi_{i,j}$ denotes the preemption overhead incurred before entering this region - i.e., the cost of resuming execution if a preemption occurred after the previous region. The WCET of task τ_i is thus given by the sum of the execution times and associated overheads of all its NPRs: $e_i = \sum_j (b_{i,j} + \xi_{i,j})$

Tasks are sorted in non-decreasing order of their periods (or equivalently, deadlines due to the implicit deadline model), such that $d_i \leq d_j$ if and only if $i \leq j$. Under this convention, a task τ_i may preempt a lower-priority task τ_j only if $i < j$.

To schedule the task set on the accelerator, we adopt the earliest-deadline-first (EDF) scheduling policy. In an ideal EDF scheduler, a job becomes ready immediately upon its release and remains so until it begins execution. At each preemption point - i.e., when a task reaches a schedulable boundary - the scheduler selects the ready job with the earliest deadline and dispatches the next non-preemptive region of that job immediately.

B. Effective Periods of Tasks in CLARE^{RT}

In a theoretical EDF scheduler, a job is considered ready immediately upon its release. However, in a practical implementation, there is a non-negligible latency between the time a job is released and when it becomes ready for execution. This delay, introduced by hardware and scheduling overheads, effectively reduces the release period of tasks.

Figure 10 illustrates the release of two consecutive jobs of the same task. While the intended release interval between these jobs is exactly p_i, the actual delay between a job's release and its readiness - caused by internal state transitions, heap management operations, and contention with other job releases - introduces a latency denoted as Δp. When the second job experiences a shorter delay ($\Delta p'' < \Delta p'$), the effective release interval p_i' becomes shorter than p_i.

We define the release time of a job as the cycle in which it enters the task release FIFO, and the ready time as the first cycle when the job is added to the scheduling heap after all required heap operations. To characterize the worst-case delay, Δp_{worst}, we conservatively assume that all potential scheduler branches are exercised at least once during the release-to-ready transition. In the worst-case scenario, the maximum possible latency due to heap operations can be denoted by the following equation.

$$\Delta p_{\text{worst}} = (2N+3)\lceil \log(N) \rceil + 3N + 4 \text{ cycles} \tag{1}$$

where N is denoted as the number of tasks in the task set.

The effective interval between two consecutive jobs can be conservatively bounded by assuming that the first job experiences the worst-case delay Δp_{worst}, while the second job incurs no delay. We use this reduced interval (i.e., p_i') as the task's effective release period in the schedulability analysis.

$$p_i' = p_i - \Delta p_{\text{worst}} \tag{2}$$

C. Model of Execution Time

We define each cycle in which the scheduler issues an instruction as a preemption point. According to the execution procedure of the CLARE$^{\text{RT}}$ accelerator described in Section III, the system performs a sequence of operations between two consecutive preemption points, assuming no preemption or resume events occur: (i) The kernel management module retrieves the instruction, accesses the corresponding metadata, and initiates the accelerator. (ii) The accelerator executes the Non-Preemptive Region (NPR). (iii) Upon completion, the kernel management module provides execution feedback. (iv) The scheduler then conducts a scheduling decision and issues the next instruction.

Subsequently, the WCET of an NPR can be represented as:

$$b_{i,j} = b_{i,j}^{\text{A}} + b_{i,j}^{\text{M}} + b_{i,j}^{\text{S}} \tag{3}$$

which represents the latency spent in execution, kernel management, and scheduling operations, respectively. We provide the modeling and analysis of each item as follows:

1) Scheduling and Preemption Operation: The latency of the scheduling and preemption occurs in the scheduler, and is the latency between receiving feedback and issuing the next instruction. This latency varies since that according to Figure 6, after receiving an instruction, the scheduler will check the release of all tasks and add to the heap if any. As a worst-case estimation, we also assume that all branches of the scheduler happen once, then we have:

$$b_{i,j}^{\text{S}} = (2N+3)\lceil \log(N) \rceil + 3N + 4 \text{ cycles} \tag{4}$$

2) Kernel Management: Following the timing analysis in Section III and considering the latency in FIFO, the kernel management will have a fixed latency of:

$$b_{i,j}^{\text{M}} = 6 \text{ cycles} \tag{5}$$

3) Performance Model on Accelerator: As described in Section III, the total execution time of one non-preemptive region can be decomposed into the sum of accelerator iterations within this region:

$$b_{i,j}^{\text{A}} = \sum_k \text{I}_{i,j}^k \tag{6}$$

For each iteration, its execution time is the maximum latency of load, compute, and store operations:

$$\text{I} = \text{MAX}(e_{\text{load}}, e_{\text{comp}}, e_{\text{store}}) \tag{7}$$

For e_{comp}, we derive the latency from the original PE array design, the latency is related to the computation amount of the tile size, the number of processing elements, and the internal data flow within the PE array:

$$e_{\text{comp}} = F(\text{TM}, \text{TK}, \text{TN}, \text{PE}) \tag{8}$$

The latency of load and store are related to the DDR access. We follow the model of [27], [28]:

$$e_{\text{load}} = C_{\text{init}} + \frac{(\text{TM} * \text{TK} + \text{TK} * \text{TN}) * \text{BPE}}{\text{BW}_{\text{in}}} \tag{9}$$

$$e_{\text{store}} = C_{\text{init}} + \frac{\text{TM} * \text{TN} * \text{BPE}}{\text{BW}_{\text{out}}} \tag{10}$$

where BPE represents the number of bytes per element. BW_{in} and BW_{out} is the communication capability of the input and output memory interface, which is related to number AXI ports, datawidth of each port, burst length of each port and the number of consecutive accesses. All of these factors are specified in the accelerator design, thus an accurate Bandwidth can be modeled. C_{init} is the initialization overhead of each memory access, which is an attribute of the DDR hardware for the platform, we use $C_{\text{init}} = 300$ cycles in this work. The $e_{\text{load}}, e_{\text{comp}}, e_{\text{store}}$ are the same for all iterations running on the same accelerator. In particular, if an iteration does not perform some operations, the corresponding latency is set to 0.

D. Model of Preemption Overhead

As demonstrated in Section III, the preemption overhead related to one preemption point is determined by the strategy of recompute or persist. In this work, we denote that for each NPR ($\delta_{i,j}$), we use a binary variable ($d_{i,j}$) to encode the design choice of the preemption point **after** $\delta_{i,j}$, where $d_{i,j} = 0$ represents that recompute is used, and $d_{i,j} = 1$ represents persist strategy.

When a task τ_i preempts τ_k after $\delta_{k,j}$, a preemption overhead $\xi^{\text{pre}}_{k,j}$ is going to be paid, then when τ_k resumes, a resume overhead $\xi^{\text{res}}_{k,j}$ is going to be paid before executing $\delta_{k,j+1}$. Following the timing analysis in Section III, when using the recompute strategy, the preemption overhead is cleaning the output buffer, and the resume overhead is recomputing the cleaned iterations. When using the persist strategy, the preemption overhead is storing the output buffer, and the resume overhead is loading the stored output buffer, together with loading one input buffer:

$$\xi^{\text{pre}}_{i,j} = \{ \begin{array}{r} \frac{\text{TM}*\text{TN}}{\text{P}}, \text{if } d_{i,j} = 0 \\ e_{\text{store}}, \text{if } d_{i,j} = 1 \end{array} \quad (11)$$

$$\xi^{\text{res}}_{i,j} = \{ \begin{array}{r} \text{I} \times \mathsf{MAX}(e_{\text{load}}, e_{\text{comp}}), \text{if } d_{i,j} = 0 \\ e_{\text{store}} + e_{\text{load}}, \text{if } d_{i,j} = 1 \end{array} \quad (12)$$

where $\text{I} = \left(\text{cur\_I} - \frac{\text{K}}{\text{TK}} \cdot \left\lfloor \frac{\text{cur\_I}-2}{\text{K/TK}} \right\rfloor - 1\right)$, which is the number of iterations that needs to be recomputed, and cur_I stands for the current iteration index of the preemption point, which is related to the actual workload.

We then discuss how to bound the preemption overhead $\xi_{i,j}$ according to $\xi^{\text{pre}}_{i,j}$ and $\xi^{\text{res}}_{i,j}$. Note that $\xi_{i,j}$ stands for the preemption overhead that is before the NPR $\delta_{i,j}$. Suppose that task τ_i preempts τ_k after $\delta_{k,j}$, $\xi_{i,1}$ accounts for the preemption overhead $\xi^{\text{pre}}_{k,j}$, while $\xi_{k,j+1}$ accounts for the resume overhead $\xi^{\text{res}}_{k,j}$.

For the first region of task τ_i, it can only preempt other tasks with a longer period, thus its preemption overhead will be the maximum of all $\xi^{\text{pre}}_{k,j}$ that $k > i$:

$$\xi_{i,1} = \mathsf{MAX}(\xi^{\text{pre}}_{k,j}), \text{for all } k > i \quad (13)$$

The rest of the regions in each task can only be resumed. Though multiple tasks can preempt before a region, the resume overheads are only related to the PP before this region:

$$\xi_{i,j} = \xi^{\text{res}}_{i,j}, \text{for } j > 1 \quad (14)$$

TABLE I: Hardware design parameters.

Paramter	Description	Value
TM	tile size	1536
TK	tile size	128
TN	tile size	1024
#PE	PE number	384
Eff.	PE efficiency	0.8
BPE	#bytes per element	4
C_init	initialization overhead of DDR	300
BW_in	input bandwidth (byte/cycle)	84
BW_out	output bandwidth (byte/cycle)	30
BW_persist	persist bandwidth (byte/cycle)	30
BW_resume	resume bandwidth (byte/cycle)	21
N	max #tasks	15

E. Dataflow Strategy with PPP Optimization

CLARE$^{\text{RT}}$ performs schedulability analysis and PP placement following the setup in [1]–[3], using a task model where all PPs are initially enabled, as previously described. However, as shown in Equations 11 and 13, the preemption overhead for a task depends on the design strategies applied to PPs in other tasks. Consequently, all PP strategies must be determined before executing the placement algorithm. This leads to a prohibitively large design space: for a task set with n tasks and k PPs per task, there are $2^{n \cdot k}$ possible design combinations - an infeasible number to evaluate exhaustively, especially considering that intra-layer preemptive dataflows can involve hundreds of PPs in a single DNN.

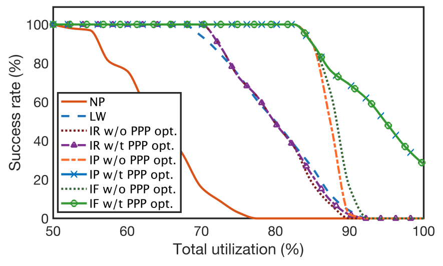
To manage this complexity, we propose a heuristic approach. From Equation 11, for any i, j, $\xi^{\text{pre}}_{i,j}$ takes one of two possible values, with e_{store} being the larger. Specifically, $\xi_{i,1} = \text{TM} \cdot \text{TN/P}$ only if every $d_{k,j}$ for $k > i$ is set to 0 (i.e., the recompute strategy is used for all such PPs). Otherwise, $\xi_{i,1} = e_{\text{store}}$. In other words, $\xi_{i,1}$ equals $\text{TM} \cdot \text{TN/P}$ only when all later tasks exclusively adopt the recompute strategy across all PPs. Based on this insight, we evaluate two heuristic PP placement strategies: (*i*) Recompute-dominant strategy: Assume all PPs use the recompute strategy (i.e., all $d_{k,j} = 0$), yielding $\xi_{i,1} = \text{TM} \cdot \text{TN/P}$, with other $\xi_{i,j}$ $(j > 1)$ derived directly from $d_{k,j}$; (*ii*) Persist-inclusive strategy: Assume at least one PP uses the persist strategy, and conservatively set $\xi_{i,1} = e_{\text{store}}$. This is safe since $\xi_{i,1} \le e_{\text{store}}$. Then, based on Equation 14, the values of $\xi_{i,j}$ for $j > 1$ depend only on the immediate preceding PP. Thus, $\xi_{i,j}(j > 1)$ is determined by $\xi^{\text{res}}_{i,j-1}$ $d_{i,j-1}$:

$$d_{i,j} = \{ \begin{array}{l} 0, \text{if I} \times \mathsf{MAX}(e_{\text{load}}, e_{\text{comp}}) < e_{\text{store}} + e_{\text{load}} \\ 1, \text{otherwise} \end{array} \quad (15)$$

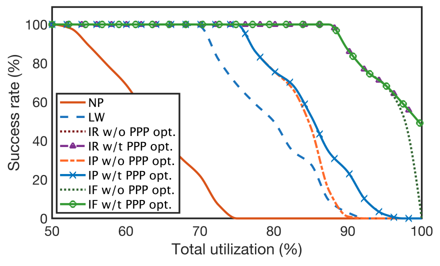
V. Evaluation

A. Experiment Setup

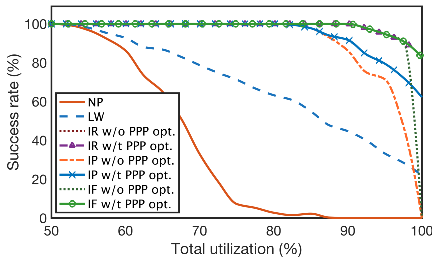
We implement the CLARE system on the AMD Versal VCK 190 platform [29] to evaluate the effectiveness of CLARE$^{\text{RT}}$. We choose CHARM [7] as the baseline accelerator with the PE array located on the AI engine of the platform. We implement a customized data buffer, memory interface, and kernel management module in the PL part to enable the intra-layer preemptive flexible dataflow. The scheduler is also implemented in the PL part. A job release module is added to replay the workload traces, and a probe using techniques



(a) Success rate on MLP1 workload.



(b) Success rate on MLP2 workload.



(c) Success rate on real-world workloads.

Fig. 11: Success rate of different preemption strategies on various workloads.

TABLE II: On-board measured latency of all accelerator operations within the system. The measurement matches the hardware design specifications. The variance is small and can be capped by the analytical performance model.

Operation	Load	Compute	Store	Recompute	Persist Preemption	Persist Resume
Avg Latency	15819	23357	204891	16385	204891	293376
Max Latency	15969	23359	204924	16388	204916	293512
Perf Model	16092	23362	210015	16400	210015	299893

in [30] is implemented to record the cycles. The design parameters used in the implementation are shown in Table I. The design on PL and AIE is compiled by Vitis 2021.1 and runs at 230 MHz and 1 GHz. AMD XRT library is used for creating the host program running on CPU.

Two synthetic workloads and five real-world workloads are evaluated. To clearly demonstrate the difference between different designs, we choose two multi-layer perceptrons (MLPs) benchmarks with a small and large hidden dimension size: Each MLP has two layers of matrix multiplications of the same shape. The layer shape of MLP 1 and MLP 2 is set to $1024 \times 8192 \times 1024$, and $2048 \times 128 \times 2048$ respectively. For the real-world workloads, we evaluate transformer and MLP-based models including DeiT-T [31], BERT-tiny [32], BERT-mini [32], PointNet [33] and MLP-Mixer [34].

B. *Effectiveness of Performance Model*

Table II demonstrates the on-board profiling results of different operations in the implemented system, which includes the latency of load, compute, store operations in execution, the preemption (cleaning output buffer) in the recompute phase, and the preemption and resume overhead in the persist phase. The cost of recomputing is combined with the load and compute operations as introduced in Equation 12. These latencies are profiled using the probe when running different applications. Besides, the proposed performance model provides a tight upper bound on the profiled latencies, which gives estimations 0.76%, 0.012%, 0.24%, 0.073%, 0.24%, and 0.21% higher than the profiled maximum value, respectively.

For the scheduling operation and kernel management operation, we use the VCS [35] to simulate the HDL codes generated by HLS to get the cycle number and compare that to the on-board execution cycle read from the hardware probes. Both on-board measured latency and simulated latency match our design specifications described in Section III-C.

C. *Effectiveness of Intra-layer Preemptive Dataflow*

1) Experiment Setup: To comprehensively compare the differences of different dataflow designs, we generate a large scale of task sets and test if they are schedulable using different dataflow, and report the success rate. For a given group of workloads where the number of tasks and the shape of each task is known, we first generate the utilization of each task based on UUniFast Algorithm [36]. The execution time of each task is obtained by the proposed performance model, then the period of each task is computed using execution time over utilization. Finally, designs with different dataflows will be tested on this task set with all designs scheduled by the EDF algorithm. At least 100 random utilizations are generated for each design at each total utilization.

A cycle-accurate simulator is implemented to conduct this large-scale exploration based on the performance model and the collected on-board profiling data of each task set. Besides, to speed up the process, we conduct a schedulability test for these design points. One design is regarded as successful if it passes the test. If a design fails the test, we then conduct a simulation to determine whether it is scheduleable.

2) Success Rate on Synthetic Workloads: Figure 11a and 11b show the success rate on task sets on synthetic workloads, where the task set in them is composed of two MLP1 workloads and two MLP2 workloads, respectively. Eight designs are evaluated: the non-preemptive dataflow (NP) and the layer-wise preemptive dataflow (LW) serve as the baseline. For the intra-layer preemptive dataflow, three strategies are provided: recompute only (IR), persist only (IP), and flexible dataflow (IF). Each intra-layer dataflow is tested before (w/o PPP) and after (w/t PPP) the PP placement.

Both figures show that the intra-layer dataflows after PP placement are consistently better than the baseline non-preemptive dataflow and the layerwise preemptive dataflow, proving the effectiveness of the intra-layer preemption. Besides, the flexible dataflow can maintain a considerable success rate higher than 50% even in a high total utilization of 95%.

Figure 11a shows the case that the workloads have a large reduced dimension (K) size. In this case, there will be only one tile in the matrix multiplication, meaning that if recomputing is used in one PP, it will recompute all the layer when resuming. As a result, the recompute dataflow, both with and without PP placement will have a similar performance as the layer-wise preemptive dataflow. In fact, the large reduced dimension size will lean toward the persist strategy, since the latency of storing and loading back the output buffer will be smaller than

TABLE III: Resource utilization of different modules within CLARE^{RT} system.

	LUT	REG	BRAM	URAM	DSP	AIE
Scheduler	9177 (1.02%)	8132 (0.45%)	1 (0.10%)	0 (0%)	0 (0%)	0 (0%)
Job Release	15135 (1.68%)	32646 (1.81%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Memory Interface	14517 (1.61%)	23218 (1.29)	0 (0%)	0 (0%)	37 (1.88%)	0 (0%)
Kernel Management	11898 (1.32%)	25053 (1.39%)	6.5 (0.67%)	0 (0%)	1 (0.05%)	0 (0%)
Accelerator	105060 (11.68%)	112324 (6.24%)	787.5 (81.44%)	384 (82.84%)	102 (5.18%)	384 (96.00%)
Total	155787 (17.31%)	201373 (11.19%)	795 (82.21%)	384 (82.94%)	140 (7.11%)	384 (96.00%)

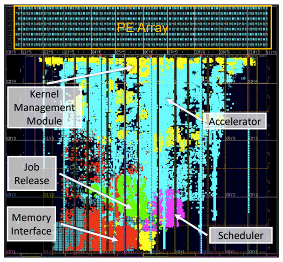


Fig. 12: System layout.

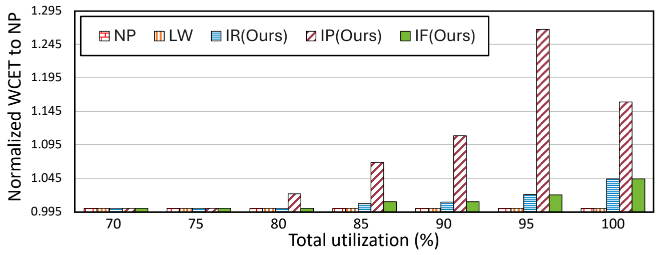


Fig. 13: Normalized WCETs of DeiT-T model.

computing a large amount of tiles in the K dimension. As a result, the persist dataflow has a better performance (IP w/t better than IR w/t, IP w/o better than IR w/o in Figure 11a). Figure 11b demonstrates another case in which the reduced dimension is small. In this case, the cost of recomputing will be smaller than persisting. Therefore, the recompute-only dataflow outperforms persist-only (IR w/t better than IP w/t, IR w/o overlaps with IF w/o, both better than IP w/o in Figure 11b).

The flexible dataflow takes advantage of both recompute and persist strategies. For the intra-layer preemptive dataflow designs without PP placement, the flexible dataflow will be better than the recompute-only and persist-only designs (Figure 11a), or at least have the same success rate as the one with the higher performance (IR w/o overlaps with IF w/o in Figure 11b). When PP placement is enabled, the flexible dataflow has the same rate as persist-only in Figure 11a and recompute-only in Figure 11b, forming the best overall performance.

3) Success Rate on Real-world Workloads: Figure 11c demonstrates the successful rates of task sets on real-world workloads. Here the number of tasks in one task set is set to 2, and the model for each task is randomly selected from the five real-world workloads. Here, layer-wise preemption is better than synthetic workload. This is because the real-world workloads have more layers, enabling more inter-layer preemption points. As a result, the layer-wise-preemptive can leverage these preemption points. Still, as the intra-layer preemptive dataflow enables more preemption points than the layer-wise dataflow, the intra-layer-preemptive flexible design after PP placement shows the best among all, especially at the high total utilization when larger than 95%.

4) Comparison of WCET of Different Preemption Strategies.: Figure 13 shows the normalized WCETs of a synthetic workload having two layers with the shape of $6144 \times 512 \times 4096$, in which the M,K,N dimensions are all partitioned into four tiles. To measure the WCETs under each utilization, we set task sets with three tasks of the model with a random distribution of utilization. The WCETs are averaged through different task sets with the same total utilization. The WCETs of the non-preemptive, layer-wise-preemptive, and three intra-layer preemptive dataflows after PP placement are shown in Figure 13. For the two baseline designs, the normalized WCETs are always the same as the execution length since there is no preemption overhead, however, the limited number of preemption points makes them miss the deadline more easily, especially when the total utilization is high. For the intra-layer preemptive designs, the overhead is also zero when utilization is low, since the design can leverage the inter-layer PPs. As the utilization increases, more PPs are enabled to ensure meeting the deadline, bringing preemption overheads. The overhead after PP placement is small compared with the total execution length, with at most 4.3% of the execution time in this case for the flexible dataflow.

Regarding the overhead of intra-layer preemptive designs before PP placement, since there are a large number of preemption points, the total overhead of the recompute-only, persist-only, and flexible dataflows are $4.03\times$,$1.29\times$, and $4.03\times$ the execution length.

D. Resource Utilization

The layout of CLARE^{RT} implementation on the VCK 190 board is shown in Figure 12, where the modules of job release, scheduler, kernel management, memory interface, and accelerator are highlighted. The resource utilization of each module is shown in Table III. The overhead brought by CLARE^{RT} is small. To enable the low-latency deterministic scheduling and the intra-layer preemptive flexible dataflow design, the additional resource usage is 5.63% LUT, 4.94% registers, 0.77% BRAM, and 1.93% DSP.

VI. Conclusion

In this paper, we propose CLARE^{RT}, a deterministic cycle-level accurate accelerator to address the limited preemption capabilities of the specialized AI accelerators and enhance the real-time performance. CLARE^{RT} enables fine-grained, intra-layer flexible preemptive scheduling with cycle-level determinism by adopting an on-chip EDF scheduler, which minimizes scheduling latency and variance, along with a customized dataflow architecture that supports intra-layer preemption with low overhead. We conduct a comprehensive predictability analysis of CLARE^{RT}, facilitating formal schedulability verification and preemption point placement optimizations under limited preemptive scheduling constraints.

References

[1] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption Points Placement for Sporadic Task Sets," in *2010 22nd Euromicro Conference on Real-Time Systems*, 2010, pp. 251–260.

[2] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal Selection of Preemption Points to Minimize Preemption Overhead," in *2011 23rd Euromicro Conference on Real-Time Systems*, 2011, pp. 217–227.

[3] B. Standaert, F. Raadia, M. Sudvarg, S. Baruah, T. Chantem, N. Fisher, and C. Gill, "A Limited-Preemption Scheduling Model Inspired by Security Considerations," 2024.

[4] M. Rafie, "Autonomous Vehicles Drive AI Advances for Edge Computing," https://www.3dincites.com/2021/07/autonomous-vehicles-drive-ai-advances-for-edge-computing/.

[5] Y. Huang, J. Chen, and D. Huang, "Ufpmp-det: Toward accurate and efficient object detection on drone imagery," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, no. 1, 2022, pp. 1026–1033.

[6] Y. Nagamatsu, F. Sugai, K. Okada, and M. Inaba, "Basic implementation of FPGA-GPU dual SoC hybrid architecture for low-latency multi-DOF robot motion control," in *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2020, pp. 7255–7260.

[7] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, D. Chen, J. Cong, and P. Zhou, "CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 153–164. [Online]. Available: https://doi.org/10.1145/3543622.3573210

[8] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[9] S. Liu, S. Yao, X. Fu, R. Tabish, S. Yu, A. Bansal, H. Yun, L. Sha, and T. Abdelzaher, "On removing algorithmic priority inversion from mission-critical machine inference pipelines," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 319–332.

[10] ——, "Taming algorithmic priority inversion in mission-critical perception pipelines," *Communications of the ACM*, vol. 67, no. 2, pp. 110–117, 2024.

[11] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "Timewall: Enabling time partitioning for real-time multicore+ accelerator platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 455–468.

[12] S. Kim, H. Genc, V. V. Nikiforov, K. Asanović, B. Nikolić, and Y. S. Shao, "MoCA: Memory-centric, adaptive execution for multi-tenant deep neural networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 828–841.

[13] S. Zeng, G. Dai, N. Zhang, X. Yang, H. Zhang, Z. Zhu, H. Yang, and Y. Wang, "Serving multi-DNN workloads on FPGAs: A coordinated architecture, scheduling, and mapping perspective," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1314–1328, 2022.

[14] C. Gao, Y. Wang, C. Liu, M. Wang, W. Chen, Y. Han, and L. Zhang, "Layer-Puzzle: Allocating and Scheduling Multi-task on Multi-core NPUs by Using Layer Heterogeneity," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[15] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 584–597.

[16] C. Wang, Y. Bai, and D. Sun, "CD-MSA: cooperative and deadline-aware scheduling for efficient multi-tenancy on DNN accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 7, pp. 2091–2106, 2023.

[17] F. Restuccia and A. Biondi, "Time-predictable acceleration of deep neural networks on fpga soc platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 441–454.

[18] AMD, "AMD Vitis™ AI Software." [Online]. Available: https://www.amd.com/en/products/software/vitis-ai.html

[19] S. Ji, X. Chen, W. Zhang, Z. Yang, J. Zhuang, S. Schultz, Y. Song, J. Hu, A. K. Jones, Z. Dong *et al.*, "Towards accelerator customization in real-time safety-critical systems," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025, pp. 181–181.

[20] J. Guan, R. Wei, D. You, Y. Wang, R. Yang, H. Wang, and Z. Jiang, "Mesc: Re-thinking algorithmic priority and/or criticality inversions for heterogeneous mcss," in *2024 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2024, pp. 1–14.

[21] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems," in *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 2010, pp. 352–355.

[22] E. Rossi, M. Damschen, L. Bauer, G. Buttazzo, and J. Henkel, "Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing With FPGAs," vol. 11, no. 2, 2018. [Online]. Available: https://doi.org/10.1145/3182183

[23] S. Attia and V. Betz, "Feel free to interrupt: Safe task stopping to enable FPGA checkpointing and context switching," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 1, pp. 1–27, 2020.

[24] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 1–12.

[25] M. Pagani, A. Balsini, A. Biondi, M. Marinoni, and G. Buttazzo, "A linux-based support for developing real-time applications on heterogeneous platforms with dynamic fpga reconfiguration," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017, pp. 96–101.

[26] B. Seyoum, M. Pagani, A. Biondi, and G. Buttazzo, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 481–490.

[27] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 105–115.

[28] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls. in 2017 54th acm/edac/ieee design automation conference (dac)," 2017.

[29] AMD/Xilinx, Versal AI Core Series VCK190 Evaluation Kit.

[30] Y.-k. Choi and J. Cong, "Hlscope: High-level performance debugging for fpga designs," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 125–128.

[31] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International conference on machine learning*. PMLR, 2021, pp. 10 347–10 357.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[33] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.

[34] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "MLP-mixer: an all-MLP architecture for vision," in *Proceedings of the 35th International Conference on Neural Information Processing Systems*, ser. NIPS '21. Red Hook, NY, USA: Curran Associates Inc., 2024.

[35] Synopsys, "VCS: Functional Verification Solution." [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[36] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, 2005.