RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling

Sudhanshu Gupta
University of Rochester
sgupta45@cs.rochester.edu

Sandhya Dwarkadas
University of Virginia
sandhya@virginia.edu

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00084

***Abstract*—Data movement latency when using on-chip accelerators in emerging heterogeneous architectures is a serious performance bottleneck. While hardware/software mechanisms such as peer-to-peer DMA between producer/consumer accelerators allow bypassing main memory and significantly reduce main memory contention, schedulers in both the hardware and software domains remain oblivious to their presence. Instead, most contemporary schedulers tend to be deadline-driven, with improved utilization and/or throughput serving as secondary or co-primary goals. This lack of focus on data communication will only worsen execution times as accelerator latencies reduce.**

In this paper, we present RELIEF (RElaxing Least-laxIty to Enable Forwarding), an online least laxity-driven accelerator scheduling policy that relieves memory pressure in accelerator-rich architectures via data movement-aware scheduling. RELIEF leverages laxity (time margin to a deadline) to opportunistically utilize available hardware data forwarding mechanisms while minimizing quality-of-service (QoS) degradation and unfairness. RELIEF achieves up to 50% more forwards compared to state-of-the-art policies, reducing main memory traffic and energy consumption by up to 32% and 18%, respectively. At the same time, RELIEF meets 14% more task deadlines on average and reduces worst-case deadline violation by 14%, highlighting QoS and fairness improvements.

I. Introduction

Modern smartphone systems-on-chip (SoCs) comprise of several dozens of domain-specific hardware accelerators dedicated to processing audio, video, and sensor data [42]. These accelerators, which sit outside the CPU pipeline, are referred to as *loosely-coupled accelerators* (LCAs). They appear as programmable I/O devices to the OS and communicate with the CPU using memory-mapped registers and shared main memory, sometimes connecting to the last-level cache [16]. To maximize performance and accelerator-level parallelism [43], applications can request a chain of accelerators running in producer/consumer fashion [38]. The speedups these chains provide, however, is limited by the fact that the accelerators communicate via the main memory, creating contention at the memory controller and the interconnect. This bottleneck will worsen as SoCs become more heterogeneous and incorporate accelerators for more elementary operations [15].

Techniques to reduce this contention include 1) *forwarding* data from the producer to the consumer, i.e., moving data from the producer's local memory directly to the consumer's, and 2) *colocation* of consumer tasks with producer tasks, thus eliminating all data movement. Examples of *forwarding* techniques include insertion of intermediate buffers between producer and consumer accelerators (VIP [38], [58]) or optimizing the cache coherence protocol to proactively move data from producer's cache to the consumer's cache directly (FUSION [28]). The former requires design-time determination of communicating accelerator pairs, while the latter requires that the accelerators use caches and be part of the same coherence domain, limiting their scalability and flexibility. More recent techniques include ARM AXI-stream [4], [6], which allows multiple producer/consumer buffers to be connected over a crossbar switch, and Linux P2PDMA [31], [50], which enables direct DMA transfers between PCIe devices without intermediate main memory accesses. Unlike VIP and FUSION, they allow for dynamic creation of producer/consumer pairs at run time in order to move data between them. Efficient utilization of such *forwarding* techniques, however, remains a challenge.

Existing systems expect software to explicitly utilize the forwarding mechanism to move data between producer and consumer [36], [38], [40], requiring knowledge of task mapping to accelerators. Distributed management of tasks by each accelerator, however, results in the accelerator's inability to utilize forwarding mechanisms due to the lack of knowledge of task mappings to other accelerators. A centralized accelerator manager has a global view of the system, allowing implementation of policies that opportunistically employ forwarding mechanisms to improve accelerator utilization and application performance. Unfortunately, the scheduling policies employed thus far [15], [20] by these managers are not designed to efficiently utilize forwarding hardware.

Scheduling policies typically prioritize tasks using arrival time, deadline, or laxity. Such policies can be extended to prioritize tasks that may forward data from a producer, similar to FR-FCFS scheduling in memory systems [46], where row buffer hits are prioritized over older tasks. However, this can lead to unfairness where an application with more forwards can starve others with fewer forwards. Therefore, **we need a scheduling policy that can opportunistically perform data forwards while still providing fairness and quality of service (QoS)**.

In this paper, we introduce RELIEF, an online accelerator scheduling policy that has forwarding, QoS, and fairness as first-class design principles. RELIEF prioritizes newly ready tasks over existing ones since they can move data directly from the producer's memory using forwarding mechanisms. RELIEF provides QoS in terms of meeting task deadlines and

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00084

fairness in terms of reducing variance in application slowdown due to contention. It achieves both by tracking task laxity and throttling priority elevations if they can cause missed deadlines. These properties matter where tail-latency is important, such as user-in-the-loop smartphone and client-server applications. We evaluate RELIEF on a suite of vision and machine learning benchmarks with strict latency constraints on a mobile platform. Our key contributions are:

- An evaluation of data movement overheads for low-latency accelerator chains used in deadline-constrained vision and machine learning applications. We observe that some of these applications spend as much as 75% of their execution time on data movement.
- A novel scheduling policy, called RELIEF, that maximizes utilization of existing forwarding hardware. RELIEF can be easily integrated into existing hardware managers and is agnostic of both the forwarding mechanism and the specific definition of laxity, allowing for wider adoption.
- Extensive evaluation of RELIEF on a simulated mobile SoC, encompassing performance improvements, implementation overheads, and sensitivity to microarchitectural design decisions. RELIEF achieves up to 50% more forwards compared to state-of-the-art (SOTA) policies, resulting in 32% and 18% lower main memory traffic and energy consumption, respectively. Simultaneously, RELIEF improves QoS by meeting 14% more task deadlines on average, and improves fairness by reducing the worst-case deadline violation by 14%.

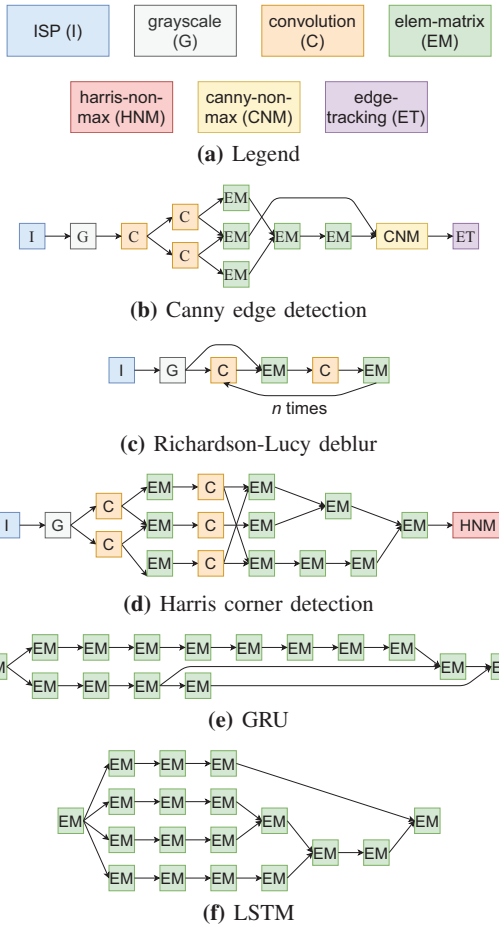
II. Background

General-purpose processors and domain-specific accelerators represent two ends of a spectrum of performance and flexibility, with the latter trading off the former's versatility for improved performance. A middle ground between the two approaches is to have a set of accelerators for elementary operations that can be stitched together dynamically by each application to serve its needs [15]. This is supported by the observation that applications across domains are often composed of similar kernels [22]. Such an approach eliminates redundancy of hardware functional units along with greatly minimizing the need for a specialized accelerator for each new application.

In this section, we present a suite of real-time smartphone workloads that are widely used in modern devices and discuss how they can be broken down into a set of elementary accelerators. We quantify how memory-bound these accelerators are, motivating the need for techniques to reduce data movement costs. Next, we discuss the functionality of an accelerator manager [15] and why they are well-equipped to improve hardware utilization and provide QoS. Finally, we present examples to explain how SOTA accelerator scheduling policies fall short in utilizing forwarding hardware.

A. Modern smartphone workloads

We study two important classes of modern smartphone workloads in this paper: vision and recurrent neural networks.



(a) Legend

(b) Canny edge detection

(c) Richardson-Lucy deblur

(d) Harris corner detection

(e) GRU

(f) LSTM

Fig. 1: Kernels in different image processing and RNN applications

Both classes together represent a wide variety of compute-intensive user-facing applications, making them suitable for hardware acceleration.

Computer vision: Mobile visual computing applications have exploded in popularity in recent years, ranging from complex photography algorithms to AR/VR applications [19]. These applications often utilize several common image processing kernels. One example is *Canny edge detection* [10], which is used in face detection, either alone [35] or as part of a neural network pipeline [54]. Another example is *Harris corner detection* [24], which is used for feature extraction in panorama stitching algorithms [30], especially in VR applications [32]. *Richardson-Lucy deconvolution* [33], [45] is an image deblurring algorithm that sharpens shaky camera images. These three applications are commonly fed images directly from an image signal processor (ISP) that captures raw camera output and performs preprocessing operations like demosaicing, color correction, and gamma correction [25].

Recurrent neural networks (RNNs): These are a class of machine learning kernels used for time-series data, wherein the inference at a time step can affect the inference at a later time. This makes them particularly well-suited for speech

recognition [41] and language translation [55] applications in modern phones. We evaluate two different RNN applications: *long short-term memory (LSTM)* [26] and *gated recurrent unit (GRU)* [13]. Given their widespread use, RNNs have been the subject of prior work in low-latency accelerator design [21] and accelerator scheduling [14], [59].

Details about these benchmarks, including their deadline and input size, are listed in Table V. These applications can be represented as directed acyclic graphs (DAGs) of seven compute kernels, each of which can be implemented as a separate hardware accelerator, as shown in Figure 1. The description of each accelerator is listed in Table I. These accelerators are ultra low-latency, spending significant time moving data to/from memory. The data movement overhead for each accelerator and each application is quantified in Table II. For each application, the table compares the memory time without forwarding hardware to an ideal scenario where forwarding hardware is used whenever possible.

TABLE I: Elementary accelerators

Accelerator (SPAD size in B)	Description
canny-non-max (262,144)	Suppress pixels that likely don't belong to edges.
convolution (196,708)	Convolution with a max. filter size of 5x5.
edge-tracking (98,432)	Mark and boost edge pixels based on a threshold.
elem-matrix (262,144)	Element-wise matrix operations including add, mult, sqr, sqrt, atan2, tanh, and sigmoid.
grayscale (180,224)	Convert RGB image to grayscale.
harris-non-max (196,608)	Enhance maximal corner values in 3x3 grids and suppress others.
ISP (115,204)	Perform demosaicing, color correction, and gamma correction on raw images.

The percentage of time spent on data movement by each accelerator is primarily a function of its operational intensity. Accelerators like `convolution` have abundant data reuse, which leads to high operational intensity and a higher compute-to-memory access time ratio. Meanwhile, `elem-matrix` has little to no data reuse depending on the operation requested, which causes its run time to be dominated by memory access latency. The frequency of use of each accelerator type dictates how much time each application spends on data movement. GRU and LSTM, which exclusively use `elem-matrix`, spend nearly 75% of their run time moving data between accelerators while Deblur, which relies heavily on `convolution`, spends a mere 3%. More importantly, we can see how efficient use of forwarding hardware can significantly reduce data movement overheads, especially for memory heavy RNN applications.

B. Accelerator manager

The use of dedicated hardware to manage the execution of accelerators frees up the host cores from performing scheduling and serving frequent interrupts from accelerators [15], especially for applications with thousands of low latency nodes. [1] The manager implements a runtime consisting of a host interface, a scheduler, and driver functions for each accelerator type.

[1]We use the terms node and task interchangeably.

TABLE II: Absolute time spent in compute vs data movement. These are sum totals and do not account for computation/communication overlap.

Accelerator	Time (us)	
	Compute	Memory
canny-non-max	443.02	30.45
convolution	1545.61	18.25
edge-tracking	324.73	13.56
elem-matrix	10.94	30.44
grayscale	10.26	15.23
harris-non-max	105.01	13.77
ISP	34.88	8.71

Application	Time (us)		
	Compute	Mem (no fwd)	Mem (ideal)
canny	3539.37	237.74	173.29
deblur	15610.58	509.80	420.06
gru	1249.31	3343.72	1608.01
harris	6157.30	372.19	303.16
lstm	1470.02	3879.98	1797.77

Host interface: The CPU and the hardware manager communicate via shared main memory, with user programs submitting tasks to the manager via either a system call or user-space command queues [29], [44].

Scheduler: The submitted tasks are written into queues in the main memory that can be read directly by the hardware manager. The hardware manager performs sorted insertion of these tasks into their respective accelerator's ready queue using a scheduling policy. These policies typically sort using arrival time, deadline, or laxity.

Drivers: Tasks from ready queues are then launched onto accelerators via driver functions. Drivers manipulate accelerators or their DMA engine's memory-mapped registers (MMRs) to launch computations or load/store data, respectively.

Hardware managers can be realized as an accelerator themselves or as a microcontroller, with the latter trading off latency for ease of implementation and flexibility [20].

C. Limitations of SOTA scheduling policies

To illustrate how contemporary accelerator scheduling policies underutilize forwarding mechanisms, consider the two DAGs presented in Figure 2a. The number of each accelerator type available is indicated in the "Accelerators" box. The color inside each node represents the type of resource it requires. The upper number is the execution time of the node while the lower number is the deadline. The node deadlines have been computed using critical-path method assuming both DAGs arrive at time 0 and have deadlines of 16 and 15 time units, respectively.

We compare the schedules generated by four SOTA policies to an ideal schedule. Each of the policies presented below work by sorting a per accelerator-type ready queue based on the described criteria. As an accelerator of a given type becomes available, the manager runtime pops the head of the queue for execution.

1) *First Come First Serve (FCFS)*: Simplest baseline policy where incoming tasks are appended to the tail of the

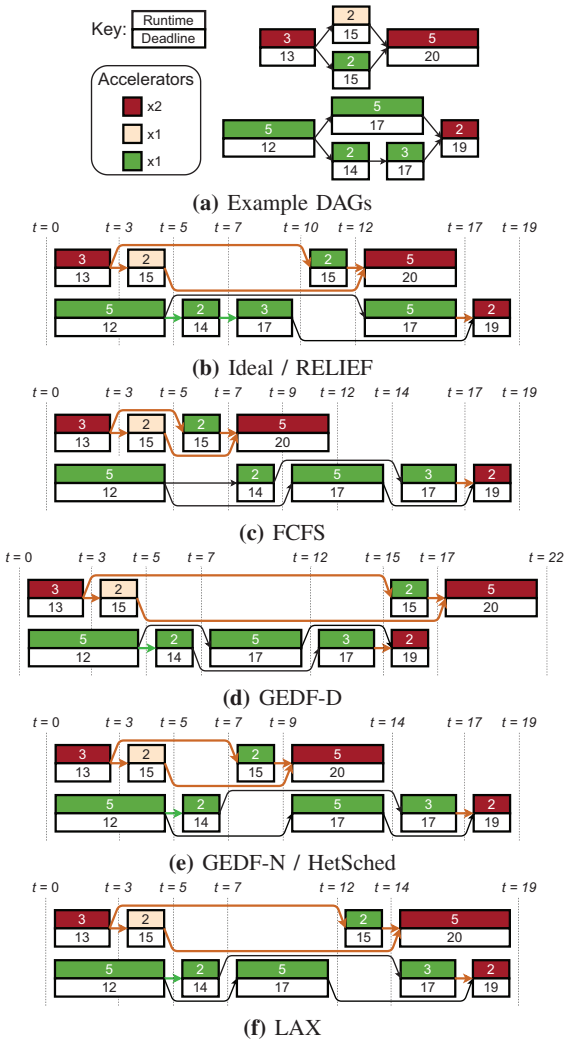


Fig. 2: Comparison of FCFS, GEDF-D, GEDF-N, LAX, and HetSched to an ideal schedule. RELIEF achieves the ideal schedule. Brown and green arrows represent forwarding and colocation, respectively. The "Accelerators" box indicates the available number of each accelerator type.

ready queue. FCFS represents the non-preemptive version of round-robin scheduling used in GAM+ [15].

2) *Global Earliest Deadline First (GEDF)*: A straightforward extension of the uniprocessor optimal Earliest Deadline First (EDF) policy, where the tasks are sorted based on increasing deadline. There are two variants depending on how the task deadlines are computed:
   a) *GEDF-DAG (GEDF-D)*: Uses the deadline of the DAG that the task belongs to as the task deadline. This was previously used in VIP [38].
   b) *GEDF-Node (GEDF-N)*: Sets the task deadline by performing critical-path analysis on the DAG. This is amongst the most well-studied policies in real-time literature [51], [56], [60].
3) *Least Laxity First (LL)*: Another uniprocessor optimal scheme [17], this policy works by sorting tasks in increasing order of their *laxity*, which is defined below in Equation 1. The deadline used here is set using the critical-path method.

$$laxity = deadline - runtime - current\_time \quad (1)$$

4) *LAX [59]*: A variant of LL that de-prioritizes tasks with a negative laxity in favor of tasks with a non-negative laxity to improve the number of tasks that meet their deadline. We use this variant of LL for comparison in the rest of the paper.
5) *HetSched [3]*: A least-laxity first policy that assigns task deadlines using the following equation:

$$deadline_{task} = SDR \times deadline_{DAG} \quad (2)$$

Here, *sub-deadline ratio (SDR)* quantifies the contribution of a task to the execution time of the path it is on.

Figure 2 shows the possible schedules generated by each of the policies above. The figures shows both cases, one where data is forwarded from producer to consumer (brown arrow), and another when computation is colocated, putting the consumer computation on the same accelerator as the producer, thereby eliminating all data movement (green arrow). For the same number of forwards, the policy that generates more colocations is therefore the better one. Note that intermediate results are dispensable; we only care about the final output. Looking at the ideal schedule in Figure 2b, we observe that it not only meets deadlines, but also achieves 5 forwards and 2 colocations. The ideal policy is able to achieve these forwards and colocations by running the consumer nodes immediately after producer nodes, allowing for better utilization of the aforementioned forwarding techniques. To the best of our knowledge, this is an optimization that no current scheduling policy performs. All other policies, barring GEDF-D, meet the deadline but miss out on forwarding opportunities. FCFS achieves 5 forwards, but performs no colocations. GEDF-D achieves better forwarding with 5 forwards and 1 colocation but misses deadlines. GEDF-N and HetSched produce the same schedule where they meet deadlines but with sub-optimal number of colocations. LAX has a different schedule than GEDF-N/HetSched, but achieves the same number of forwards. We, therefore, need a scheduling policy that exploits forwarding opportunities while being deadline aware.

III. RELIEF: Relaxing Least-Laxity to Enable Forwarding

A. Scheduling algorithm

We now present RELIEF, *RElaxing Least-laxIty to Enable Forwarding*, our proposed LL-based policy that attempts to maximize the number of data forwards while delivering QoS. The key idea behind the policy is to promote nodes whose parents have just finished execution, ensuring that the children can forward the data from the producer before it is overwritten. To reduce unfairness and missed deadlines such promotions might cause, RELIEF employs a laxity-driven approach that

throttles priority escalations when deadlines could potentially be missed. By combining priority elevations with laxity-driven throttling, RELIEF achieves the ideal schedule shown in Figure 2b as well as the ideal data movement time in Table II. We can see from the figure how RELIEF's behavior deviates from LAX, another LL-based policy, at timestep 7, where RELIEF favors the second DAG's newly ready child over existing ready nodes with lower laxity and deadlines.

The RELIEF algorithm is presented in Algorithm 1. Newly ready nodes whose parents have just finished execution are called *forwarding nodes*, since they can potentially forward data from the producer's local memory. RELIEF schedules these forwarding nodes immediately if there are resources available, bypassing nodes with lower laxity if they can meet their deadline under a LL scheme. If no priority escalation is possible, the algorithm proceeds in a vanilla LL fashion. We also experiment with LAX's de-prioritization mechanism that allows tasks with non-negative laxity to bypass those with negative laxity in the ready queue (Section II-C). While this mechanism can improve the number of tasks that complete by their deadline (Section V-D), we show that it can lead to unfairness in Section V-E.

Algorithm 1: RELIEF

```
1  Function RELIEF (finishing_node):
2      for child ∈ finishing_node.children do
3          child.cmplt_parents += 1
4          if child.cmplt_parents == child.num_parents then
5              child.runtime = predict_runtime(child)
6              child.laxity = child.deadline - child.runtime
7              index = find_pos(fwd_nodes[child.acc_id], child)
8              fwd_nodes[child.acc_id].insert(index, child)
9      for each acc_id do
10         max_forwards = num_idle_accelerators[acc_id]
11
12         while not fwd_nodes[acc_id].empty() do
13             node = fwd_nodes[acc_id].pop_front()
14             index = find_pos(ready_queue[acc_id], node)
15
16             if max_forwards > 0 and
                 is_feasible(ready_queue[acc_id], node, index)
                 then
17                 ready_queue[acc_id].push_front(node)
18                 node.is_fwd = true
19                 max_forwards -= 1
20                 update_fwd_metadata(finishing_node, child)
21             else
22                 ready_queue[acc_id].insert(index, node)
23                 node.is_fwd = false
```

RELIEF works by creating a laxity-sorted list of candidate forwarding nodes, called *fwd_nodes*, from newly ready nodes (Algorithm 1, lines 2-8). We store laxity as *deadline - runtime*, subtracting the current time from it when manipulating the ready queue (Algorithm 2, line 6). The candidate nodes are then inserted into the ready queue at either the front (Algorithm 1, line 17) or at the position dictated by their laxity (Algorithm 1, line 22). A candidate node is escalated in priority only if 1) the number of forwarding nodes in the ready queue for an accelerator type is less than the number of idle instances of that type (controlled by `max_forwards`), and 2) the function `is_feasible()` returns true. The first condition ensures that forwarding nodes are always the next to run, ensuring their input data is still live in its producer's local memory. `is_feasible()` returns true if the priority escalation of the candidate node is unlikely to cause deadline misses. Our evaluation shows that predicting node *runtime* once at the time of insertion into the ready queue has sufficient accuracy (Section V-F).

The key to minimizing missed deadlines is `is_feasible()`'s ability to predict which node promotions might cause them. It takes three arguments: the ready queue, the candidate forwarding node, and its position in the ready queue based on laxity. In our implementation, presented in Algorithm 2, we use a laxity-driven approach. For each node in the ready queue that has a higher priority than the candidate node, we ensure that its laxity is more than the candidate node's run time. That is, each of those nodes can tolerate the additional latency of the candidate node without missing their deadline. Since the queue is already sorted by laxity, we start at the head of the queue and find the first node that is 1) itself not a forwarding node, and 2) has positive laxity. If the node thus found has laxity greater than the candidate node's runtime, then every following node does too and the candidate node's priority can be safely escalated. The first condition here ensures that existing forwarding nodes do not prevent escalation of other nodes, while the second is an optimization that lets us bypass negative laxity nodes since they are not expected to meet their deadlines even without the promotion.

Algorithm 2: is_feasible

```
1  Function is_feasible(ready_queue, fnode, index):
2      can_forward = True;
3      for node ∈ ready_queue do
4          if ready_queue.index(node) == index then
5              break;
6          curr_laxity = node.laxity - curTick();
7          if not node.is_fwd and curr_laxity > 0 then
8              can_forward = curr_laxity > fnode.runtime;
9              break;
10     if can_forward then
11         for node ∈ ready_queue do
12             if ready_queue.index(node) == index then
13                 break;
14             node.laxity -= fnode.runtime;
15     return can_forward
```

B. Execution time prediction

Since RELIEF and its feasibility check are laxity-driven, they require an estimate of each node's execution time. We accomplish that by predicting the compute time and memory access time of each task separately.

Compute time prediction: The compute time of fixed-function accelerators, such as the ones used in this study, is largely a function of the input size and the requested computation, owing to the data-independent nature of their control flow [14]. The compute time of such devices can, therefore, be profiled just once at either design time or system boot-up since there will be very little variation. Our evaluation shows that this scheme has an average error of just 0.03% (Section V-F).

Memory time prediction: The memory access time prediction works by predicting two values: the available bandwidth and the amount of data movement. For the former, we experiment with three different predictors based on prior work [18]: *Last value*, *Average*, which computes the arithmetic mean of the bandwidth of n previous tasks, and *Exponentially Weighted Moving Average (EWMA)*, that computes a weighted sum of the most recently achieved bandwidth (bw) and historical data, as shown below:

$$pred_n = \alpha \times bw + (1 - \alpha) \times pred_{n-1} \tag{3}$$

The data movement predictor works by analyzing the graph and observing node states. For predicting input data movement, we need to predict if a node can be colocated with its parent, since colocations eliminate producer/consumer data movement. Given that the scheduler performs colocations by tracking the previously executed node on an accelerator, only one child can be colocated. We predict that the child with the earliest deadline of a set of newly ready children will colocate with the parent if they use the same accelerator type.

For predicting output data movement, we need to predict the number of forwards. If all children can forward from the node, then we will not need to write results back to the main memory. This will be true if a) all the children map to a unique accelerator, and b) all the children will be ready when the node finishes. The former is a simple comparison between the number of tasks mapping to an accelerator type and the instances of that type, while the latter is achieved by ensuring that the node is the latest finishing parent based on its deadline.

The accuracy and performance of bandwidth predictors compared to a *Max* prediction scheme, where the maximum available bandwidth is used, are presented in Section V-F. We also compare the data movement predictor to a *Max* prediction scheme where maximum data movement is assumed.

C. System architecture

We present the system architecture that we assume in Figure 3. The accelerators are modeled to directly access physical memory without address translation, like some existing designs [39]. We propose exposing the entire scratchpad memory in each accelerator to the rest of the system via a non-coherent read-only port. The newly exposed scratchpad memories are not mapped to user address space and access is hidden behind device drivers, ensuring secure access. We also use a discrete hardware manager that is coherent with the CPUs (Section II-B), responsible for scheduling nodes onto accelerators as well as for orchestration of data movement between producers and consumers.

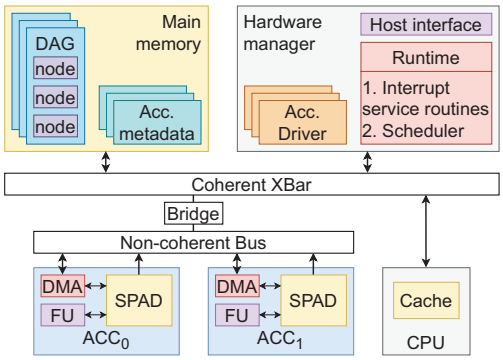


Fig. 3: System architecture depicting the hardware manager and the interconnect.

The CPUs, the hardware manager, and the accelerators communicate via shared main memory and interrupts. The CPU informs the hardware manager of new DAGs by writing the root nodes into shared queues in the main memory. Each node is a structure that represents a task for an accelerator, as shown in Table III. The hardware manager parses each node to push them onto ready queues, and launches them on the accelerators via driver functions. The accelerators inform the manager of the completion of each task by raising an interrupt. When a node completes, the manager updates its `status` field to inform the host CPU program of its completion and pushes its children onto ready queues if their dependencies are satisfied. The user program can learn of the completion of an entire DAG by reading the `status` of leaf nodes.

TABLE III: DAG node data structure

struct node
`uint32_t acc_id;`
`void *acc_inputs[NUM_INPUTS];`
`node *children[NUM_CHILDREN];`
`node *parents[NUM_INPUTS];`
`uint8_t status;`
`uint32_t deadline;`
`acc_state *producer_acc[NUM_INPUTS];`
`uint32_t producer_spm[NUM_INPUTS];`
`uint32_t completed_parents;`

The node structure contains a few more synchronization and bookkeeping fields that we hide for brevity. The size of the structure depends on the number of parents and children each node has, along with the pointer size. Assuming 32-bit pointers, the base size of the structure with a single parent and child is 72 bytes, with each additional parent and child adding 12 bytes and 4 bytes, respectively. The largest node we see in our applications is 96 bytes. While we show the arrays to be of a constant size, this implementation choice may be replaced with dynamic structures.

1) Forwarding mechanism: Exposing accelerator private scratchpad memories onto the system interconnect allows consumer DMA engines to perform reads from producer scratchpads without having to go to the main memory. Such

a modification should be fairly straightforward in modern SoCs [52], exposing the scratchpad memories to the system interconnect on the DMA plane. This is what we assume in our evaluation. It also possible to leverage PCIe resizable-BAR support [2], which enables exposure of multiple gigabytes of private accelerator memory into the CPU address space, and Linux P2PDMA interface [31], [50], which allows for direct DMA transfers between PCIe devices.

2) Hardware manager: We now detail the data structures maintained and runtime executed by the hardware manager described in Section II-B. We chose a microcontroller-based implementation for our work since it offers sufficient performance (Section V-G).

Manager data structures: The hardware manager maintains metadata for each accelerator to track its state and to manage synchronization of data between producers and consumers. Table IV presents the key metadata fields. In addition to maintaining the address for accelerator and DMA engine MMRs (`acc_mmr` and `dma_mmr`), the metadata also holds the address of the scratchpad memory partitions (`spm_addr`), the state of the accelerator (`status`, e.g., idle or running), and the number of accelerators currently reading from each of its scratchpad partitions (`ongoing_reads`). Scratchpad partitions are used to implement multi-buffering.

TABLE IV: Accelerator metadata

struct acc_state
`uint8_t *acc_mmr;`
`uint8_t *dma_mmr;`
`uint8_t *spm_addr[NUM_SPM_PARTITIONS];`
`uint8_t status;`
`node *output[NUM_SPM_PARTITIONS];`
`uint32_t ongoing_reads[NUM_SPM_PARTITIONS];`

The scratchpad partition addresses are physical addresses used by consumer DMA engines to perform direct data transfers. The field `ongoing_reads` is used to keep track of how many consumers are reading from a scratchpad partition of the accelerator to avoid overwriting the data. The manager increments the count before a consumer starts transferring the data to its local scratchpad memory and reduces the count after it is done, thus ensuring that write-after-read dependencies are respected when data is being forwarded.

The metadata size for each accelerator in our implementation, assuming 32-bit pointers and a maximum of 3 scratchpad partitions (`NUM_SPM_PARTITIONS`), is 32 bytes, totaling to 236 bytes for the 7 accelerators our system simulates.

Manager runtime: Alongside launching tasks onto accelerators, the manager runtime implements an interrupt service routine (ISR) and the scheduler. The ISR is triggered every time an accelerator finishes a *job*, where a job could be a DMA operation or computation.

Once an accelerator finishes execution and the scheduler is run, the field `output[p]` (Table IV) is set to point to the node that just finished, denoting that partition p holds the node's output. The `producer_acc` and `producer_spm` fields are also set in the child nodes to inform their drivers of which producer accelerator and partition to read from. When child nodes are launched, their driver checks if the data is still present in the producer's scratchpad and forwards it if it is. In addition, if all the child nodes are not at the head of their respective ready queue (i.e., not next in line for execution), or the parent node does not have any children, the runtime calls the producer driver to write the results back to main memory immediately.

IV. Evaluation Methodology

A. Benchmarks

We evaluate RELIEF against the four policies summarized in Section II using three vision and two RNN applications. The five applications, along with their input size, deadline, and laxity (when run alone), are listed in Table V. We assume the vision applications run at 60 frames per second (FPS) and thus use a deadline of 16.6 ms. Deadline for RNN applications has been borrowed from previous work [59]. Input sizes mirror prior work as well [15], [59]. Richardson-Lucy deblur is an iterative algorithm where higher iterations lead to better picture quality. We use 5 iterations to have a representative input size balanced with simulation time. Along similar lines, we assume a sequence length of 8 for both LSTM and GRU.

TABLE V: Benchmarks

(Symbol) Benchmark	**Input / hidden layer size**	**Deadline**	**Laxity**
(C) Canny edge detection [10]	128 x 128	16.6 ms	13.6 ms
(D) RL deblur [33], [45]	128 x128	16.6 ms	0.2 ms
(G) GRU [13]	128	7 ms	2.3 ms
(H) Harris corner detection [24]	128 x 128	16.6 ms	14 ms
(L) LSTM [26]	128	7 ms	3.6 ms

B. Platform

We use gem5-SALAM [47] for our evaluation, which provides a cycle-accurate model for accelerators described in high-level C. The simulator consumes the description of an accelerator in LLVM [1] intermediate representation (IR) and a configuration file and provides statistics like execution time and energy consumption. These accelerators are then mapped into the simulated platform's physical address space, enabling access via memory-mapped registers. The simulated configuration, listed in Table VI, models a typical mobile device [38]. We model the hardware manager using an ARM Cortex-A7 based microcontroller running bare-metal C code. Cortex-A7 has an area and power overhead of 0.45mm^2 and $<$100mW [5], which can be reduced further by stripping the vector unit. The simulated platform models end-to-end execution of applications, from inserting the tasks into ready queues till the completion of each requested application. This includes interrupt handling, scheduling, driver functionality, DMA transfers, and accelerator execution. In addition to the bus-based interconnect between the accelerators listed in Table VI, we evaluate RELIEF's performance with a crossbar switch in Section V-H. The two topologies represent two ends of the interconnect cost/performance spectrum.

Our evaluation uses seven image processing accelerators, one each for the kernels shown in Figure 1. Each accelerator was

TABLE VI: Simulation setup

Hardware manager	ARM Cortex-A7 based 1.6 GHz single-core in-order CPU 32 KB 2-way L1-I; 32 KB 4-way L1-D; 64 B cache line size
Main memory	LPDDR5-6400; 1 16-bit channel; 1 rank; BG mode; t_{CK} = 1.25ns; burst length = 32 Peak bandwidth = 12.8 GB/s
Interconnect	Full-duplex bus; width = 16 B Peak bandwidth = 14.9 GB/s

designed in isolation by determining the *energy*×*delay*^{2} (*ED*^{2}) product for the execution of a single task on the accelerator, while varying the configuration in terms of the number of functional units and memory ports. The configuration with the minimum *ED*^{2} was chosen for the design, similar to previous work [47], [53]. In practice, we expect accelerators to work on the same input size to allow for easy chaining and sharing of data by commonly used applications. Our accelerators, clocked at 1 GHz, thus, have enough scratchpad memory to work on 128x128 inputs along with double buffered output to avoid blocking on consumer accelerator reads. The precise scratchpad memory sizes are listed in Table I. For accelerators with differing input sizes, the software runtime or the hardware manager can break down tasks into smaller chunks, similar to accelerator composition in GAM+ [15].

C. System load

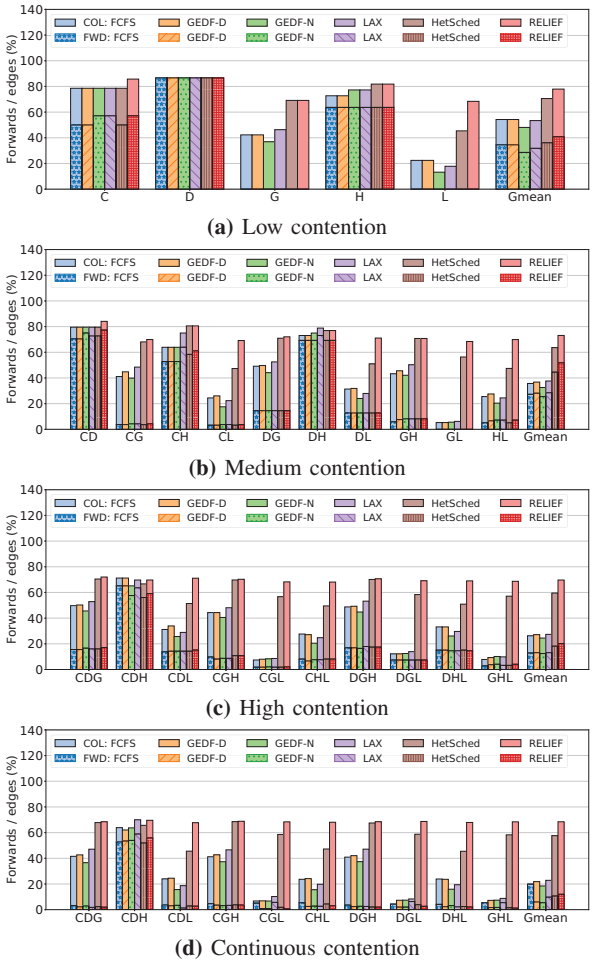
Combinations of the applications in Table V are often seen in real-world scenarios, e.g., Canny+LSTM is used for lane detection in self driving cars [57]. Enumerating all combinations of these applications, thus, helps us cover all their existing and potential future use cases. We experiment with four levels of contention to see how each of the policies scale. *Low* contention is just a single application, *medium* contention is all combinations of size 2, while *high* contention is all combinations of size 3. Increasing contention represents reduced ability to meet deadlines, with combinations larger than 3 meeting very few deadlines and thus not evaluated. In each of these scenarios, each application is instantiated once and the simulation ends when the last application finishes execution. The fourth level of contention, called *continuous* contention, is a modification of *high* contention where each of the three applications are run in a continuous loop to ensure each application experiences contention throughout its execution. We limit the execution time of each simulation to 50ms and report results for finished tasks. Each application is represented with a symbol in the following figures, as listed in Table V.

V. Results

A. Data forwards

Our primary design goal with RELIEF is producing more data forwards than SOTA policies. We quantify this increase in Figure 4.

Observation 1: SOTA policies under-utilize forwarding mechanisms. In contrast, RELIEF consistently achieves >65% of all possible forwards, on average. This is clear from Figure 4, which shows the percentage of total data forwards and colocations, computed as the ratio of number of forwards/colocations to the total number of edges in the mix. We can see how SOTA policies' obliviousness to data forwarding mechanisms leads to their under-utilization, achieving as little as 8% of all forwards possible. In contrast, RELIEF improves over HetSched, the leading SOTA policy, by nearly 1.2x on average under continuous contention.



(a) Low contention

(b) Medium contention

(c) High contention

(d) Continuous contention

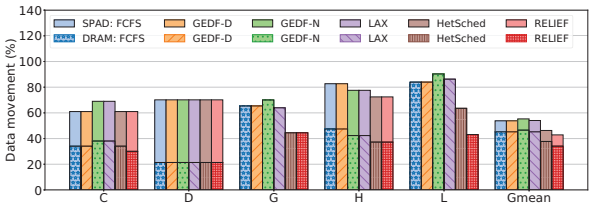
Fig. 4: Percent of total forwards and colocations, computed as the ratio of the total number of forwards/colocations to the total number of edges in the mix.

We observe two trends across all three four of contention in Figure 4: 1) RNN applications (GRU and LSTM) are the biggest contributors to colocations, and 2) application mixes with more RNN applications achieve better forwarding with RELIEF than others. The first observation is unsurprising given that all RNN tasks map onto a single resource. For the second observation, we attribute the gains with RNN applications to the fact that they contain long, linear chains (up to 9 nodes) that have the same structure and node deadlines. Having the same node deadlines means that deadline-aware policies schedule each

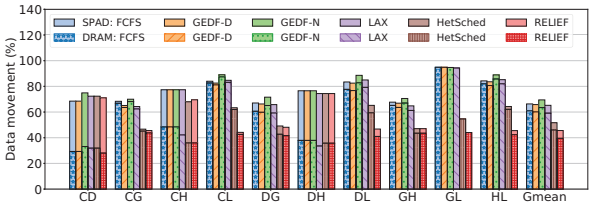
of those chains in a round-robin fashion, thus forfeiting any forwarding opportunities. FCFS has a similar problem of being locality oblivious. HetSched is able to achieve significantly more forwards than other baseline policies. These gains stem primarily from HetSched's ability to prioritize GRU's critical path, which happens to contain most of its forwards.

B. Data movement

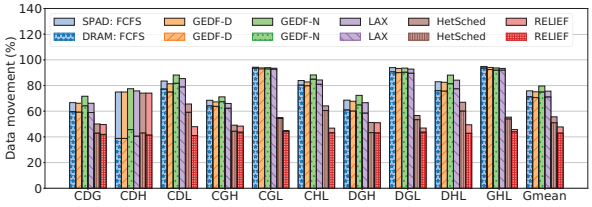
To understand each policy's data movement behavior, Figure 5 plots the percentage of data transfers (in bytes) that materialize as main memory accesses, scratchpad-to-scratchpad transfers, and colocations.



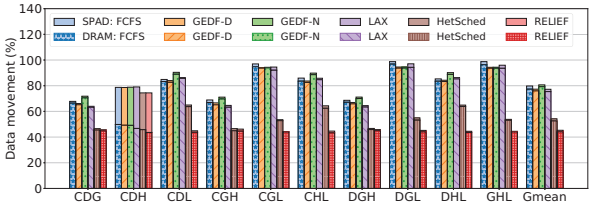
(a) Low contention



(b) Medium contention



(c) High contention



(d) Continuous contention

Fig. 5: Breakdown of data movement into main memory traffic (lower bars), SPAD-to-SPAD traffic (upper bars), and colocations (empty space). Data is normalized to total data movement when all loads and stores go to main memory.

Observation 2: RELIEF reduces main memory traffic by up to 32% compared to HetSched, under each level of contention. The average reduction compared to HetSched rests at 10%, 14%, 16%, and 16%, for low, medium, high, and continuous contention, respectively. This is a key result and highlights how simple changes to the scheduler can yield significant reductions in memory traffic.

The percentage of forwards that materialize as colocations in a mix is a function of its application composition. As explained before and evident from Figure 5a, all GRU and LSTM forwards are colocations since these applications map to a single accelerator. In contrast, the vision applications are more diverse in their resource needs and exhibit a greater degree of scratchpad-to-scratchpad data movement. The behavior of single applications impacts the behavior of entire mixes. Mixes CD, CH, and DH (medium contention), for instance, have fewer colocations than other mixes. The same is true for mix CDH (high/continuous contention).

The reduction in data movement traffic reduces energy consumption for both the main memory *and* scratchpad memories. We quantify this reduction for the high contention scenario in Figure 6.

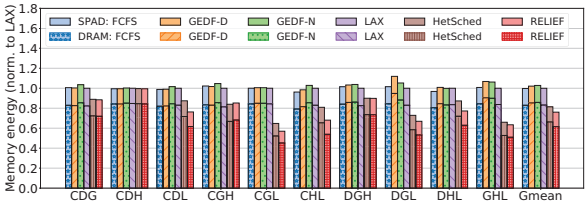


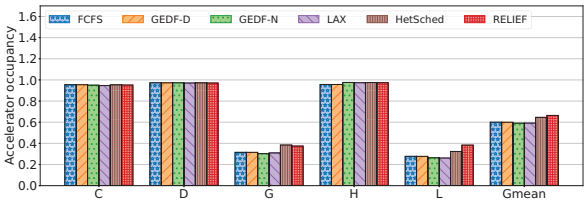
Fig. 6: Total main memory and scratchpad memories' energy consumption under high contention using gem5-SALAM's energy models.

Observation 3: RELIEF reduces main memory and scratchpad memory energy consumption by up to 18% and 8%, respectively, compared to HetSched under high contention. The average main memory and scratchpad energy reduction compared to HetSched is 7% and 4%, respectively. Forwards reduce main memory traffic while colocations eliminate both main memory and scratchpad memory traffic. While forwards cause an increase in scratchpad activity, colocations more than make up for the increase. RELIEF has the same scratchpad energy consumption as LAX for CDH, for instance, but reduces it by 24% for CGL.

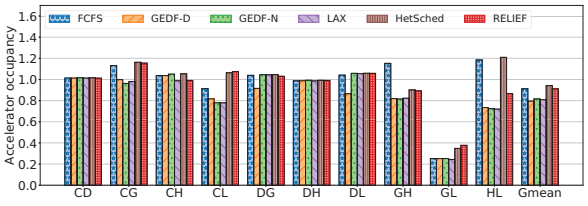
C. Accelerator utilization

Figure 7 shows accelerator utilization (or occupancy), defined as the sum, across all accelerators, of the fraction of total execution time for which each accelerator was busy. Accelerator occupancy provides a measure of degree of parallelism in each scenario. Note that while the numerator is relatively constant under the low, medium, and high contention scenarios, the denominator, which is total execution time, is impacted both by the degree of computational parallelism and by the data movement cost resulting from the use of each policy. For the continuous contention scenario, the denominator remains constant, while the numerator is impacted by the number and type of nodes executed, the data movement cost, and the degree of computational parallelism, all of which vary by policy.

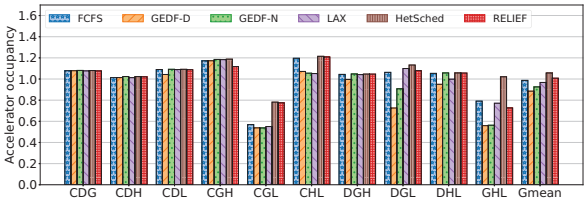
Observation 4: RELIEF improves accelerator utilization by up to 41%, compared to LAX under high contention, with an average improvement of 4%. HetSched, in turn,



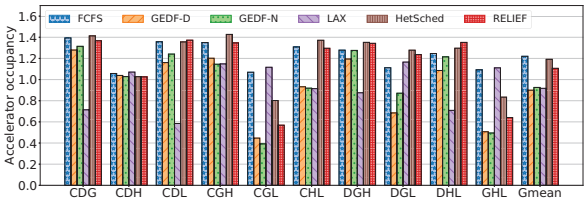
(a) Low contention



(b) Medium contention



(c) High contention



(d) Continuous contention

Fig. 7: Accelerator occupancy is defined as ratio of the sum of total of all accelerators' compute time to the the end-to-end system execution time, measured from the initiation of all applications to the completion of the last application. Higher is better.

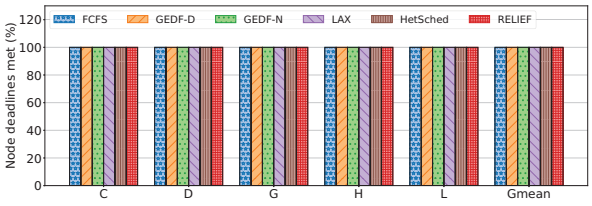
results in best case and average improvements of 41% and 5% relative to RELIEF, respectively. RELIEF's improvements over LAX are a result of increased number of forwards, resulting in lower execution time. In its attempt to increase forwards, RELIEF can sometimes hinder the progress of tasks whose children map to different accelerators, resulting in a lower degree of parallelism. This is especially evident in mixes CGL and GHL under continuous contention, where GRU and LSTM tasks, all of which map to `elem-matrix`, get promoted frequently, limiting the time they execute in parallel with the vision tasks, which utilize a variety of accelerators. HetSched and LAX's gains over RELIEF for these application mixes are primarily attributed to RELIEF's lower accelerator-level parallelism and increased scheduling latency (Section V-G).

While RELIEF's promotions reduce the degree of parallelism on average relative to HetSched, they do not cause unfairness. In fact, it is a fairer policy when compared to LAX and HetSched, as we will see in Section V-E.

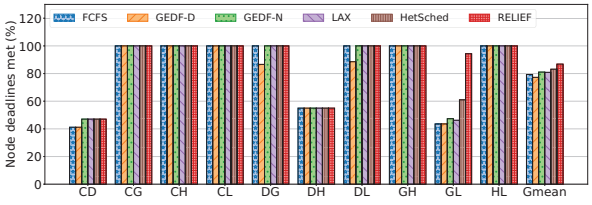
D. Node deadlines met

RELIEF integrates a feasibility check (Section III) that makes a best-effort to minimize missed deadlines. To evaluate its efficacy, we compute the percentage of node deadlines met in each application mix and present the results in Figure 8.

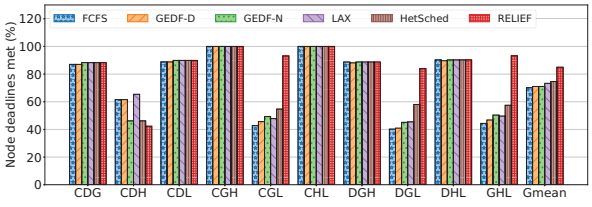
Observation 5: RELIEF meets up to 70% more node deadlines compared to HetSched, under high contention, with an average improvement of 14%. More importantly, RELIEF rarely *reduces* the number of deadlines met compared to SOTA. This highlights the effectiveness of the feasibility check in throttling priority elevations to prevent deadline violations.



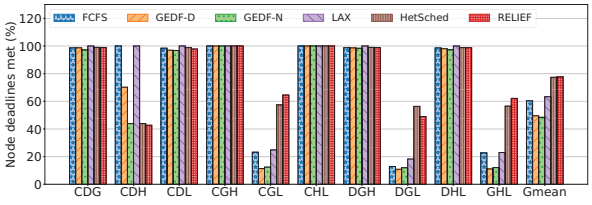
(a) Low contention



(b) Medium contention



(c) High contention



(d) Continuous contention

Fig. 8: Percent of node deadlines met

The only instance where RELIEF performs worse than existing policies is in the high contention mix CDH. We observe that GEDF-N and RELIEF prioritize Deblur nodes over Canny and Harris nodes since the former have a lower deadline and laxity. This causes nearly all of the Canny and Harris nodes to miss their deadlines. Furthermore, not all Deblur nodes meet their deadlines either because of high contention. HetSched has a similar story of prioritizing Deblur due to its longer critical path. LAX's ability to de-prioritize applications with negative

TABLE VII: Number of finished DAGs in each application mix under continuous contention.

Policy	C	D	G	C	D	H	C	D	L	C	G	H	C	G	L	C	H	L	D	G	H	D	G	L	D	H	L	G	H	L
FCFS	8	1	11	4	0	4	8	1	8	5	11	5	11	3	4	5	5	8	1	11	5	2	3	4	1	5	8	3	7	4
GEDF-D	5	1	12	3	1	2	3	2	9	5	11	4	2	4	4	3	3	9	1	11	3	1	4	4	1	3	9	4	2	4
GEDF-N	4	2	11	2	1	2	3	2	8	4	11	4	2	4	4	3	3	8	1	11	3	1	4	4	1	3	8	4	2	4
LAX	5	0	11	5	0	5	3	0	8	4	11	4	12	3	4	3	3	8	0	11	4	3	3	4	0	3	8	3	7	4
RELIEF-LAX	8	1	11	4	0	4	8	1	8	5	11	5	11	3	4	5	5	8	1	11	5	2	3	4	1	5	8	3	7	4
LL	4	2	11	2	1	2	3	2	8	4	11	4	2	4	4	3	3	8	1	11	3	1	4	4	1	3	8	4	1	4
HetSched	6	1	14	2	1	2	6	1	10	6	14	5	6	7	5	6	5	10	1	14	3	3	7	5	1	3	10	7	3	5
RELIEF	5	1	14	2	1	2	5	2	12	5	14	5	2	6	6	5	4	12	1	14	3	2	6	6	1	3	12	6	2	6

laxity allows it to de-prioritize Deblur, allowing all Canny and Harris nodes to make progress. FCFS does not suffer from this problem either because it does not prioritize DAGs and nodes. GEDF-D has the same schedule as FCFS given that all the DAGs in this mix have the same deadline. RELIEF also performs worse than HetSched in DGL, but the latter achieves the gains by unfairly slowing down LSTM. We will explore fairness in more detail in Section V-E.

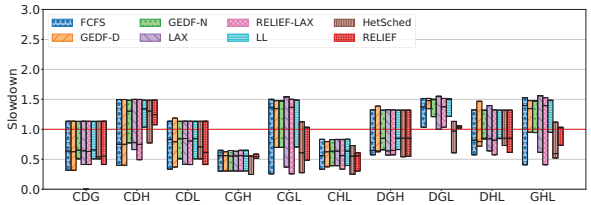
Continuous contention has a different setup compared to the other three scenarios, as described in Section IV-C. Under continuous contention, each mix executes a different number and type of nodes under different policies for a fixed period of time. In the other three scenarios, each application in a given application mix runs to completion and executes exactly once, so the number of nodes executed is constant across policies with the execution time depending on the policy's scheduling decisions. This different simulation setup results in what looks like anomalous behavior of a higher percentage of deadlines met under continuous contention compared to high contention (e.g., CDG), but in reality they cannot be directly compared. This hints at a tradeoff between deadlines met and fairness that we explore in the next section.

E. Quality-of-Service and Fairness

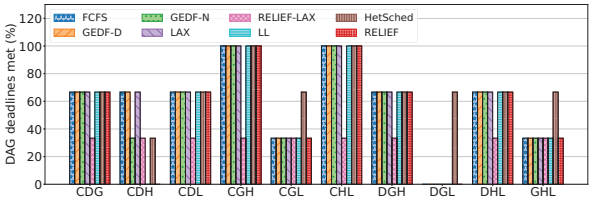
An important aspect of RELIEF's design is fairness: increased forwards for one application should not come at the cost of excessive slowdown for others. Figure 9a shows a box plot of application slowdown in each mix under high contention. The figure also shows the results for LL and *RELIEF-LAX*, a variant of RELIEF that integrates LAX's de-prioritization mechanism (Section II-C). Figure 9b, meanwhile, plots the percent of DAG deadlines met under high contention.

Figure 9a shows how RELIEF reduces maximum slowdown and variance by up to 17% and 93%, respectively, compared to HetSched. The latter meets the same or more DAG deadlines across the board, however (Figure 9b). The two results highlight a key tradeoff: HetSched meets more DAG deadlines by unfairly slowing down one application over another, as evident from its wider slowdown spread, while RELIEF attempts to distribute slowdowns and allows each DAG to make progress commensurate with its deadline. This tradeoff is made even more evident under continuous contention, as shown in Figures 10a and 10b.

Observation 6: RELIEF improves fairness, reducing worst-case deadline violation and variance by up to 14% and 98%, respectively, compared to HetSched under

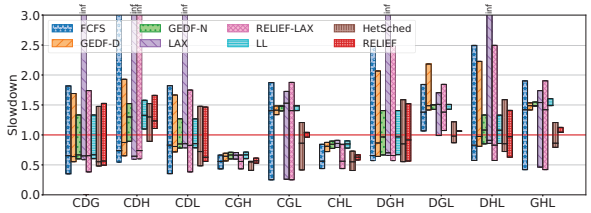


(a) Slowdown is defined as the ratio of an application's runtime to its deadline. The box edges and the median represent the slowdown for each of the three applications.

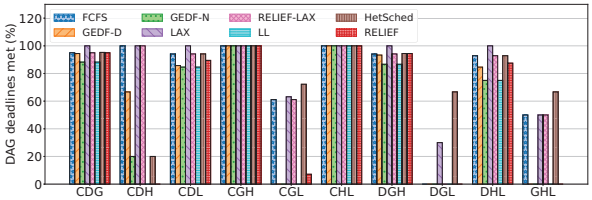


(b) Percent of DAG deadlines met.

Fig. 9: Slowdown (a) and DAG deadlines met (b) under high contention.



(a) Slowdown is defined as the ratio of an application's runtime to its deadline. The box edges and the median represent the geometric mean slowdown for each of the three applications. Infinite values represent starved applications.



(b) Percent of DAG deadlines met.

Fig. 10: Slowdown (a) and DAG deadlines met (b) under continuous contention.

TABLE VIII: Accuracy of compute time and data movement predictors, along with the accuracy and performance of memory bandwidth predictors. Negative error values represent underestimation of true value while positive error values represent overestimation. The geometric mean uses absolute error values.

Mix	Compute error (%)	Memory DM error (%)	Memory BW error (%)				Forwards				Node deadlines met			
			Max	Last	Average	EWMA	Max	Last	Average	EWMA	Max	Last	Average	EWMA
CDG	0.06	-0.95	-56.33	5.85	-1.24	1.1	139	138	138	139	136	136	136	136
CDH	0	-8.06	-59.03	-19.42	-3.95	-4.68	46	46	47	47	22	22	22	22
CDL	-0.05	-0.88	-56.47	5.19	-1.27	2.02	155	155	155	155	160	160	160	160
CGH	0.1	-1.01	-55.7	7.13	-1.18	2.19	130	130	130	130	150	150	150	150
CGL	0.02	0.59	-55.39	11.23	0.42	4.37	230	230	232	231	257	255	254	252
CHL	0.05	-0.93	-56.63	5.93	-0.64	2.79	143	143	143	143	174	174	174	174
DGH	0.03	-3.14	-56.94	4.26	-1.33	0.96	142	142	142	142	142	142	142	142
DGL	-0.02	-2.15	-55.5	8.95	-0.07	2.67	244	245	244	245	240	242	239	242
DHL	0	-3.33	-56.7	3.65	-1.36	1.31	156	156	157	157	166	166	166	166
GHL	-0.05	-0.57	-55.41	11.13	0.09	3.06	237	238	239	238	263	261	260	258
Gmean	0.03	1.47	56.4	7.31	0.68	2.22	-	-	-	-	-	-	-	-

continuous contention. HetSched is able to meet more DAG deadlines (Figure 10b) and improve accelerator utilization (Section V-C) by unfairly favoring some applications over others. For instance, HetSched meets 10 DAG deadlines in DGL while RELIEF meets 0, but it does so by slowing down one application (LSTM) by 22%. In contrast, every application suffers a slowdown of <7% under RELIEF , accompanied by a 98% reduction in variance.

We also see how LAX's de-prioritization mechanism causes significant unfairness in mixes CGL, DGL, and GHL. In all three cases, the RNN applications start missing deadlines early on due to contention and are de-prioritized by LAX and RELIEF-LAX in favor of the vision applications, causing significant unfairness. This is especially troublesome considering that they have lower deadlines compared to vision applications (Table V). In contrast, RELIEF allows the RNN applications to progress alongside the vision applications, ensuring more deadlines are met while reducing unfairness.

LAX also has a starvation problem, as is made evident from Figure 10a and Table VII. The table lists the number of completed DAG iterations for each application in each continuous contention mix. We see how Deblur is starved in every mix it is in except DGL. Deblur is extremely sensitive to queuing delays given its laxity of just 0.2ms (Table V). Combined with its linear task graph, this means that if even a single Deblur node is delayed by more than 0.2ms, the node's laxity will drop below 0 and it will get deprioritized by LAX. This is precisely what happens when Deblur contends with other vision applications for the `convolution` accelerator: if any node is launched on the `convolution` accelerator while a Deblur node is waiting, the latter will be stalled for at least 1.5ms (Table II), causing starvation. This stalls any progress for Deblur until the system has no other node to offload to the `convolution` accelerator. DGL does not suffer from this problem because GRU and LSTM do not use the `convolution` accelerator. FCFS also has 0 finished Deblur iterations in CDH, but our experiments show that it is not starved; rather it is making slow progress.

F. Prediction accuracy

The feasibility check presented in Section III utilizes a predictor to estimate compute and memory access times for accelerators. Table VIII presents the error in the compute time, the data movement, and the different memory bandwidth predictors under high contention, along with the latter's impact on the number of forwards and node deadlines met. We empirically chose *n=15* for *Average* and $\alpha = 0.25$ for *EWMA* for the best accuracy.

Observation 7: Compute time prediction has a maximum error of just 0.03%. This validates prior observations that compute time can be defined as a function of input size and requested operation for fixed function accelerators [14].

Data movement prediction also works well, with an average error of 1.35%. Memory bandwidth predictors, meanwhile, exhibit a range of accuracies, with *Average* performing the best both in terms of mean (0.68%) and maximum (3.95%) error. Their accuracy has little to no impact on performance, however. We can see from Table VIII how each policy achieves essentially the same number of forwards and deadlines met.

To understand the incremental impact of data movement and memory bandwidth predictors, Figure 11 plots the performance impact of the two predictors in isolation and combined, normalized to having *Max* predictor for both. The bandwidth predictor here is *Average*. We can see how little impact the accuracy of the predictor has on RELIEF's ability to meet deadlines. Their impact on forwards (not shown) is similar.

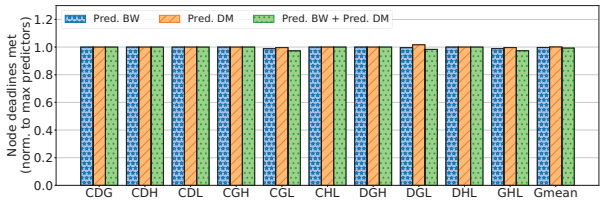


Fig. 11: Impact of memory predictors on missed deadlines under high contention.

Observation 8: RELIEF does not benefit from dynamic memory time prediction. Each application has several forwarding chains, which are contiguous sequence of forwarding producers/consumers. The laxity calculation based on the memory time prediction decides how these chains get broken up into sub-chains and interleaved. We notice that the number of sub-chains produced by each predictor does not differ

significantly, which is why they all achieve similar overall performance. Given this observation, we have used the baseline *Max* predictors for all our evaluations since they offer the same performance for negligible overhead.

G. Scheduler execution time

The execution time of a scheduling policy is an important factor in choosing one, since a better schedule may not offset the overhead of preparing the schedule itself. Figure 12 plots the average and tail latency of pushing a task into the ready queue for each policy on a Cortex-A7 based microcontroller.

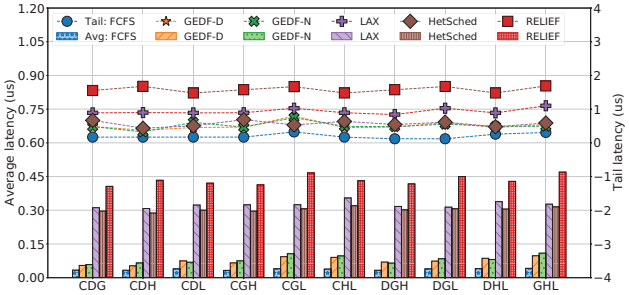


Fig. 12: Average (bars) and tail (lines) latency of the scheduler with different policies on a Cortex-A7 based microcontroller, under high contention.

Observation 9: RELIEF has higher overhead than existing policies, but is easily overlapped with accelerator execution. Looking at Figure 12 and Table II, we can see that RELIEF's modest scheduling overhead can be easily overlapped with computation, minimizing its contribution to the critical path.

H. Impact of interconnect topology

A crossbar is a high-throughput switch allowing up to $n \times m$ concurrent transactions for n requesters and m responders. This should benefit RELIEF since it permits concurrent transactions between independent producer/consumer pairs. Figure 13 shows RELIEF's sensitivity to the interconnect in terms of interconnect occupancy and the total execution time, under high contention.

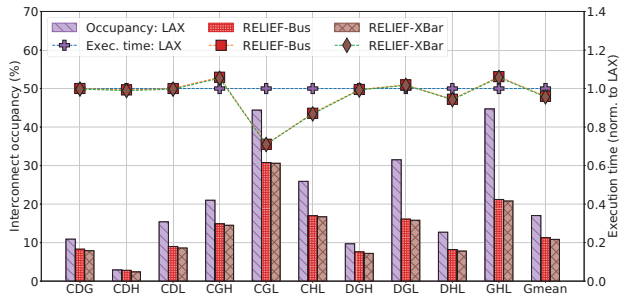


Fig. 13: RELIEF's sensitivity to system interconnect under high contention. Interconnect occupancy is defined as the percentage of cycles for which the interconnect had at least one transaction going through.

Observation 10: RELIEF reduces interconnect occupancy by up to 49% compared to LAX, with an average reduction of 33%. It does not, however, benefit from high-performance interconnects. RELIEF's low interconnect occupancy is a result of its reduction in data movement (Section V-B) as well as a lack of accelerator-level parallelism (Section V-C). This indicates that these applications are not interconnect-bound, an observation further supported by the fact that the average queuing delay for the bus is less than a cycle (not shown). We expect applications with more varied resource needs and larger input sizes to benefit more from complex interconnects.

VI. Related Work

GPU Scheduling: Prior work in GPU scheduling has looked at co-scheduling and distributing work across CPUs and GPUs to reduce synchronization and data-movement overhead [23], [34]. PTask [48] optimizes for fairness and tries to reduce data movement by scheduling child tasks onto the same device as the producer when possible. Cilk [9] also implements a child-first scheduling policy that improves locality but it optimizes primarily for improved hardware utilization. While being child-first, both PTask and Cilk are deadline blind, rendering them unsuitable for real-time applications. Zahaf et al. [60] use an EDF policy to determine which device each node should be mapped to (e.g., GPU, DSP) such that all DAG deadlines are met. Their work can be extended by optimizing for better colocation using RELIEF.

Baymax [12] and Prophet [11] use online statistical and machine learning approaches, respectively, to predict whether an accelerator can be shared by user-facing applications and throughput-oriented applications at the same time, without violating the former's QoS requirements. RELIEF can be extended with Baymax and Prophet to efficiently utilize multi-tenant accelerators like GPUs.

Menychtas et al. [37] present a fair queuing-based scheme where the OS samples each process' use of accelerators in fixed time quanta and throttles their access to provide fairness.

Accelerator scheduling: Gao et al. [21] batch identical task DAGs across multiple user-facing RNN applications together for simultaneous execution on a GPU, thereby improving GPU utilization and reducing inference latency. PREMA [14] utilizes a token-based scheduling policy for preemptive neural processing units (NPUs) that distributes tokens to each task based on its priority and the slowdown experienced due to contention, balancing fairness with QoS. While both policies are QoS-aware, neither of them optimize for data movement across multiple accelerators.

GAM+ [15] is a hardware manager that decomposes algorithms into accelerator tasks and schedules them onto physical accelerator instances using a preemptive round-robin policy. The hardware manager we used is based on GAM+. VIP [38] is an accelerator virtualization framework that uses a hardware scheduler at each accelerator to arbitrate among different applications' tasks. The authors use an EDF scheme where the FPS of the application serves as the deadline. Yeh et al. [59] propose exposure of performance counters in GPUs that drives

LAX, a non-preemptive least laxity-based scheduling policy. HetSched [3] is another laxity-driven scheduling policy for autonomous vehicles that takes task criticality and placement into account. The scheduling policies underlying these systems are used in our comparative evaluation in Section V.

Real-time scheduling: Optimal scheduling using a job-level fixed priority algorithm is provably impossible [27], unless task release times, execution times, and deadlines are known *a priori* [17]. Baruah presented optimal but NP-complete integer-linear programming formulations [7] along with approximate linear-programming relaxations [8] for scheduling real-time tasks on heterogeneous multiprocessors. Previous work also exists on providing tighter bounds on the response time of the system under both preemptive and non-preemptive variants of GEDF [51], [56]. These mathematically sound formulations provide strong performance guarantees but tend to be infeasible in an online setting. RELIEF's goal is to meet application-specified deadlines while minimizing data movement using a fast, online heuristic approach.

3DSF [49] is a hierarchical scheduler for cloud clusters that integrates three schedulers. The top layer avoids missed deadlines by using a least-laxity (LL) scheme to prioritize deadline constraint jobs over regular ones when necessary, the middle layer minimizes data movement by queuing tasks on servers that have the most inputs available locally, and the bottom layer allocates server resources to each running job proportional to its requirements. Although locality aware, 3DSF has multiple optimization targets that come with execution time overheads untenable for micro-second latency tasks.

VII. Summary and Discussion

In this paper, we present RELIEF (RElaxing Least-laxIty to Enable Forwarding), an online least laxity-based (LL-based) scheduling policy that exploits laxity to improve forwarding hardware utilization by leveraging one application's laxity to reduce data movement in another application. RELIEF increases direct data transfers between producer/consumer accelerators by up to 50% compared to SOTA, lowering main memory traffic and energy consumption by up to 32% and 18%, respectively. Simultaneously, RELIEF improves QoS by meeting 14% more task deadlines on average, and improves fairness by reducing the worst-case deadline violation by 14%. RELIEF integrates into existing architectures with hardware forwarding mechanisms and a hardware manager, requiring minimal software changes.

While we have demonstrated our ideas over LL-based scheduling, the techniques can be applied over other laxity-based policies such as HetSched as well. LL and HetSched differ in the manner in which laxity is distributed across nodes in a DAG, resulting in scheduling differences in the baseline policies. LL does not distribute its laxity, which means that each node has laxity equal to the current DAG laxity. HetSched, meanwhile, attempts to distribute the laxity among nodes based on their contribution to the critical-path execution time. With LL as a baseline policy, RELIEF has all of DAG's laxity at its disposal that it can choose to exploit whenever it sees fit. With HetSched as a baseline, however, the DAG's laxity is distributed across the nodes, limiting the number of promotions a node will allow. We are currently investigating the impact of using HetSched's laxity calculation in RELIEF. Our preliminary results indicate that such a combination continues to offer significant data movement cost savings, potentially increasing both forwards and deadlines met. We observe, however, that the choice of laxity distribution presents a tradeoff between QoS and fairness.

Acknowledgements

This work was supported in part by U.S. National Science Foundation grant CNS-1900803. We would like to thank Yuhao Zhu for his input on this work. We also thank the anonymous program committee and artifact evaluation committee reviewers for their feedback.

References

[1] "LLVM 3.8.1." [Online]. Available: https://releases.llvm.org/3.8.1/docs/index.html

[2] J. Ajanovic, "PCI Express (PCIe) 3.0 Accelerator Features." White paper: Intel.

[3] A. Amarnath, S. Pal, H. T. Kassa, A. Vega, A. Buyuktosunoglu, H. Franke, J.-D. Wellman, R. Dreslinski, and P. Bose, "Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 82–85, 2021.

[4] AMD, "AXI4-Stream Infrastructure IP Suite v3.0," May 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Infrastructure-IP-Suite-v3.0-LogiCORE-IP-Product-Guide

[5] ARM, "ARM Cortex-A7." [Online]. Available: https://developer.arm.com/Processors/Cortex-A7

[6] ARM, "AMBA AXI-Stream Protocol Specification," Apr. 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest

[7] S. Baruah, "Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms," in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 37–46.

[8] S. K. Baruah, "Partitioning real-time tasks among heterogeneous multiprocessors," in *International Conference on Parallel Processing, 2004. ICPP 2004.*, 2004, pp. 467–474 vol.1.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 207–216, event-place: Santa Barbara, California, USA. [Online]. Available: https://doi.org/10.1145/209936.209958

[10] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[11] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 17–32. [Online]. Available: https://doi.org/10.1145/3037697.3037700

[12] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 681–696. [Online]. Available: https://doi.org/10.1145/2872362.2872368

[13] K. Cho, B. v. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," 2014, _eprint: 1406.1078.

[14] Y. Choi and M. Rhu, "PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 220–233.

[15] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture Support for Domain-Specific Accelerator-Rich CMPs," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, Apr. 2014, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/2584664

[16] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[17] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[18] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 220–231.

[19] K. Fatahalian, "The Rise of Mobile Visual Computing Systems," *IEEE Pervasive Computing*, vol. 15, no. 2, pp. 8–13, Apr. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7445776/

[20] Y. Gan, Y. Qiu, L. Chen, J. Leng, and Y. Zhu, "Low-Latency Proactive Continuous Vision," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 329–342, event-place: Virtual Event, GA, USA. [Online]. Available: https://doi.org/10.1145/3410463.3414650

[21] P. Gao, L. Yu, Y. Wu, and J. Li, "Low Latency RNN Inference with Cellular Batching," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190541

[22] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[23] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems," in *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*, J. Nieh and C. A. Waldspurger, Eds. USENIX Association, 2011. [Online]. Available: https://www.usenix.org/conference/usenixatc11/pegasus-coordinated-scheduling-virtualized-accelerator-based-systems

[24] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.

[25] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 1–11, Jul. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2601097.2601174

[26] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[27] K. S. Hong and J. Y. Leung, "On-line scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 41, no. 10, pp. 1326–1331, 1992.

[28] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 733–745. [Online]. Available: https://doi.org/10.1145/2749469.2750421

[29] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," AMD, Tech. Rep., Aug. 2012.

[30] J. Li and J. Du, "Study on panoramic image stitching algorithm," in *2010 Second Pacific-Asia Conference on Circuits, Communications and System*, vol. 1, 2010, pp. 417–420.

[31] Linux Kernel Organization, "PCI Peer-to-Peer DMA Support." [Online]. Available: https://www.kernel.org/doc/html/latest/driver-api/pci/p2pdma.html

[32] Q. Liu, "Smooth Stitching Method of VR Panoramic Image Based on Improved SIFT Algorithm," in *Proceedings of the Asia Conference on Electrical, Power and Computer Engineering*, ser. EPCE '22. New York, NY, USA: Association for Computing Machinery, 2022, event-place: Shanghai, China. [Online]. Available: https://doi.org/10.1145/3529299.3531499

[33] L. B. Lucy, "An iterative technique for the rectification of observed distributions," *Astronomical Journal*, vol. 79, p. 745, Jun. 1974.

[34] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, pp. 45–55. [Online]. Available: https://doi.org/10.1145/1669112.1669121

[35] S. Madabusi and S. V. Gangashetty, "Edge detection for facial images under noisy conditions," in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, 2012, pp. 2689–2693.

[36] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, "Agile SoC Development with Open ESP," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020, event-place: Virtual Event, USA. [Online]. Available: https://doi.org/10.1145/3400302.3415753

[37] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 301–316. [Online]. Available: https://doi.org/10.1145/2541940.2541963

[38] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP Chains on Handheld Platforms," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 655–667, event-place: Portland, Oregon. [Online]. Available: https://doi.org/10.1145/2749469.2750382

[39] NVIDIA, "NVIDIA Deep Learning Accelerator (NVDLA)," 2017. [Online]. Available: www.nvdla.org

[40] NVIDIA, "CUDA Runtime API," Jun. 2023. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf

[41] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "Light Gated Recurrent Units for Speech Recognition," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 2, pp. 92–102, 2018.

[42] V. J. Reddi, "Mobile SoCs: The Wild West of Domain Specific Architectures," Sep. 2018. [Online]. Available: https://www.sigarch.org/mobile-socs/

[43] V. J. Reddi and M. D. Hill, "Accelerator-Level Parallelism (ALP)," Sep. 2019. [Online]. Available: https://www.sigarch.org/accelerator-level-parallelism/

[44] S. Rennich, "CUDA C/C++ Streams and Concurrency." [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[45] W. H. Richardson, "Bayesian-Based Iterative Method of Image Restoration," *J. Opt. Soc. Am.*, vol. 62, no. 1, pp. 55–59, Jan. 1972, publisher: OSA. [Online]. Available: http://www.osapublishing.org/abstract.cfm?URI=josa-62-1-55

[46] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 128–138, event-place: Vancouver, British Columbia, Canada. [Online]. Available: https://doi.org/10.1145/339647.339668

[47] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi, "gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 471–482.

[48] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 233–248. [Online]. Available: https://doi.org/10.1145/2043556.2043579

[49] J. Ru, Y. Yang, J. Grundy, J. Keung, and L. Hao, "An efficient deadline constrained and data locality aware dynamic scheduling framework for multitenancy clouds," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 5, p. e6037, 2021, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6037.

[Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6037
[50] M. Rybczynska, "Device-to-device memory-transfer offload with P2PDMA," Oct. 2018. [Online]. Available: https://lwn.net/Articles/767281/
[51] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel Real-Time Scheduling of DAGs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3242–3252, 2014.
[52] M. C. d. Santos, T. Jia, M. Cochet, K. Swaminathan, J. Zuckerman, P. Mantovani, D. Giri, J. J. Zhang, E. J. Loscalzo, G. Tombesi, K. Tien, N. Chandramoorthy, J.-D. Wellman, D. Brooks, G.-Y. Wei, K. Shepard, L. P. Carloni, and P. Bose, "A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022, event-place: San Diego, California. [Online]. Available: https://doi.org/10.1145/3508352.3561102
[53] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/document/7783751/
[54] R. Sikarwar, A. Agrawal, and R. S. Kushwah, "An Edge Based Efficient Method of Face Detection and Feature Extraction," in *2015 Fifth International Conference on Communication Systems and Network Technologies*, 2015, pp. 1147–1151.
[55] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112, event-place: Montreal, Canada.
[56] K. Yang, M. Yang, and J. H. Anderson, "Reducing Response-Time Bounds for DAG-Based Task Systems on Heterogeneous Multicore Platforms," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. Brest, France: Association for Computing Machinery, 2016, pp. 349–358. [Online]. Available: https://doi.org/10.1145/2997465.2997486
[57] W. Yang, X. Zhang, Q. Lei, D. Shen, P. Xiao, and Y. Huang, "Lane Position Detection Based on Long Short-Term Memory (LSTM)," *Sensors (Basel, Switzerland)*, vol. 20, no. 11, May 2020, place: Switzerland.
[58] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Short-Circuiting Memory Traffic in Handheld Platforms," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 166–177.
[59] T. T. Yeh, M. Sinclair, B. M. Beckmann, and T. G. Rogers, "Deadline-Aware Offloading for High-Throughput Accelerators," in *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.
[60] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, M. Bertogna, and G. Lipari, *A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters*. arXiv, 2019, arXiv preprint: 1901.02450.

APPENDIX

A. Abstract

This artifact appendix describes how to run RELIEF and other accelerator scheduling policies described in this paper using gem5. The artifact includes the implementation of all the policies in gem5 and our vision and RNN benchmark suite, along with pre-built binaries for the latter. It also includes optional instructions to rebuild the benchmark binaries and hardware models.

B. Artifact check-list (meta-information)

- **Algorithm:** RELIEF, a least-laxity based scheduling policy.
- **Program:** gem5 (C++ and Python code).
- **Compilation:** GCC, SCons.
- **Binary:** Vision and RNN binaries, compiled using GNU Arm Embedded toolchain v8.3.
- **Run-time environment:** Any modern Linux distribution.
- **Hardware:** X86-based CPU with 10 cores and 32GB main memory.
- **Metrics:** Data forwards, data movement, accelerator occupancy, slowdown, node deadlines met, DAG deadlines met.
- **Output:** gem5 statistics and execution trace.
- **How much disk space required (approximately)?:** 17 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2-3 hours.
- **How much time is needed to complete experiments (approximately)?:** 8-10 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Archived (provide DOI)?:** **https://doi.org/10.5281/zenodo.10237117**

C. Description

1) How to access: The code is available on GitHub[2] and Zenodo[3].

2) Hardware dependencies: Recent X86 based CPU with at least 10 cores and 32GB main memory. The simulations take multiple hours to run, and we recommend at least 60 cores and 150 GB of main memory to run all of them in parallel.

3) Software dependencies:

- Linux OS with a recent version of GCC.
- Python 2 with pip installed.
- (Optional) GNU Arm Embedded toolchain v8.3
- (Optional) LLVM 3.8

D. Installation

The steps below detail the installation for gem5 and associated Python dependencies. There is a step for building the benchmarks that requires GNU Arm Embedded toolchain v8.3. Our distribution already includes benchmark binaries, so this step is optional.

1) Navigate to the project root directory and install Python dependencies by running:
   `pip install -r requirements.txt`
2) Build gem5 by following the instructions in README.md.
3) Set `M5_PATH` environment variable:
   `export M5_PATH=`pwd``
4) (Optional) Build the benchmarks by navigating to `$M5_PATH/benchmarks/scheduler/sw` and running the following command. Note that this requires the installation of GNU Arm Embedded toolchain, described in README.md.
   `./create_binary_combinations_3.sh`
   The binaries will be put in the directory `bin_comb_3`.
5) (Optional) Compile the accelerator descriptions into LLVM IR by navigating to `$M5_PATH/benchmarks/scheduler/hw` and running `make`. Note that this requires the installation of LLVM 3.8, described in README.md.

[2] https://github.com/Sacusa/gem5-SALAM/tree/HPCA_2024
[3] https://doi.org/10.5281/zenodo.10237117

E. Experiment workflow

Navigate to the project root directory and launch high contention scenario simulations by running:

```
./run_combinations_3.sh `nproc`
```

The simulations need at least 10 cores to finish in a reasonable period. The results will be saved in the directory `$M5_PATH/BM_ARM_OUT/comb_3`.

F. Evaluation and expected results

We provide five scripts in `$M5_PATH/BM_ARM_OUT/scripts/comb_3`: `plot_forwards.py`, `plot_data_movement.py`, `plot_accelerator_occupancy.py`, `plot_slowdown.py`, and `plot_deadlines_met.py`, that reproduce Figures 4c, 5c, 7c, 9a, and 8c, respectively. The last script also reproduces 9b. Each script can be run as:

```
python <script>
```

The scripts use matplotlib to produce the figures in PDF format, which are stored in `$M5_PATH/BM_ARM_OUT/scripts/plots`.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html