CALM: Contention-Aware AcceLerator Architecture for Multi-Tenancy Execution

Seah Kim
UC Berkeley

Avinash Nandakumar
UC Berkeley

***Abstract*—Driven by the wide adoption of deep neural networks (DNNs) across many different application domains, multi-tenancy execution, where multiple DNNs are deployed simultaneously on the same hardware, has been proposed to satisfy the latency requirements of different applications while improving the overall system utilization. However, multi-tenancy execution could lead to undesired system-level resource contention, causing quality-of-service (QoS) degradation, especially for latency-critical applications. In addition, the interference caused by co-located workloads can be difficult to predict in advance due to complex SoC architectures and variable workload compositions.**

To address this challenge, we propose CALM, a light-weight, contention-aware multi-tenancy architecture for DNN accelerators. Unlike existing solutions that statically partition on-chip shared resources across applications, CALM dynamically manages the contentiousness of co-located applications to ensure the QoS target of the workloads. Specifically, CALM leverages the regularities in both DNN operators and accelerators to dynamically manipulate memory access rates based on their latency targets so that the co-located applications get the resources they currently demand without significantly starving their co-runners. We demonstrate that CALM can increase the service-level agreement (SLA) satisfaction rate up to $8.6\times$ ($2.3\times$ overall) while decreasing the performance variation by up to $37.8\times$ ($2.9\times$ overall). Our evaluation also shows that CALM is capable of improving both the system throughput by $1.11\times$ and fairness by $1.38\times$ while improving the performance variation by $8\times$.

I. Introduction

Recent advances in deep learning have led to broad adoption of DNN-based algorithms in tasks ranging from object detection [1], natural-language processing [2], AR/VR [3]–[5], and robotics [6], [7]. As a result, DNNs have become a core building block for a diverse set of applications running on edge devices, automotive processors, and all the way to data centers, many of which need to be executed concurrently to meet their latency requirements [8], [9]. To support concurrent execution of these applications, *multi-tenancy* execution, where multiple applications can be co-located on the piece of hardware, is required to improve the overall system efficiency.

The challenge associated with supporting effective multi-tenancy execution in DNN accelerators lies in the performance variability caused by contention for shared resources, especially in the case of user-facing, latency-sensitive applications that require a predictable and strictly-defined QoS to meet their latency requirements. Specifically, while the compute array in DNN accelerators is highly modular that can be easily partitioned across applications, the sharing of the memory subsystem, such as cache capacity and DRAM bandwidth, can be challenging due to the lack of coordination across applications at the system level.

The cross-application interference in traditional general-purpose cores have been well-studied in the computer architecture community, where contention-aware hardware-software scheduling [10]–[13] and code transformation [14]–[17] have been proposed to improve co-location efficiency. In contrast, today's accelerator development has been focusing on improving the efficiency of training and inference for individual applications, without any multi-tenancy support [18], resulting in disallowing co-location of DNN workloads, because of its low hardware utilization and reduced throughput [19]–[21].

More recently, new techniques to temporally interleave the execution of multiple DNNs [22]–[24] or spatially partition the local resources such as processing elements and scratchpads [5], [25] have been proposed to support multi-tenancy execution in DNN accelerators. While these techniques are effective in managing sharing of accelerator local resources, little attention has been paid to the efficient sharing of the memory subsystem, e.g., the shared system cache and DRAM, where multiple accelerators and general-purpose cores could compete for bandwidth and capacity, leading to high performance variability across runs due to unpredictable system-level interference. As the computing industry dives further into the era of domain-specific computing with a growing number of accelerators, there is a clear need for efficient contention-management mechanisms in accelerator design that can adaptively adjust the system-level contentiousness, making it possible to exploit accelerator-level parallelism of future-generation SoCs [26].

To address this challenge, we present CALM, a contention-aware accelerator architecture to support multi-tenancy execution. In contrast to prior work that focuses on efficient sharing of local resources of accelerators, CALM aims to manage the system-level contention of shared resources adaptively by dynamically manipulating the contentiousness of co-located applications. In particular, CALM leverages the regularity in DNN operators and hardware where the execution latency is highly correlated with the number of in-flight memory requests. As a result, based on a user-specified QoS target, CALM can adaptively modulate the memory access rates of co-located applications so that they can collaboratively meet their latency targets without unnecessarily starving their co-runners.

CALM consists of 1) a light-weight hardware interference monitoring and regulation engine and 2) an intelligent runtime system that manipulates the contentiousness based on user-specified adaptation policies. Our evaluation demonstrates that

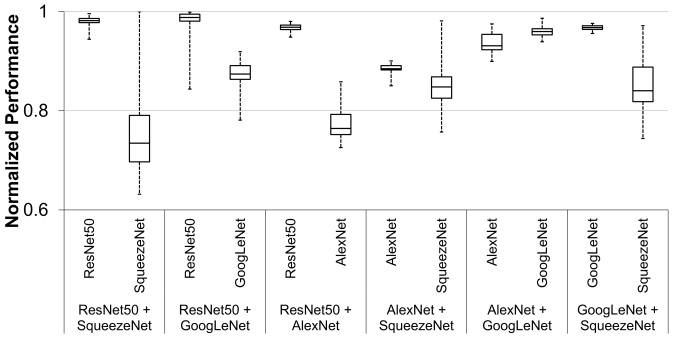


Fig. 1: Distribution of performance degradation caused by system-level interference. Performance is normalized to the latency of their isolated execution without interference. (The box is drawn from the 25th to the 75th percentile of the dataset with a horizontal line in the middle denoting the median.)

CALM can increase SLA satisfaction rate by up to $8.6\times$ ($2.3\times$ overall) while reducing the performance variation by $37.8\times$ ($2.9\times$ overall) in addition to improving both system throughput by $1.11\times$ and fairness by $1.38\times$ while improving variation by $8\times$.

In summary, this paper makes the following contributions:

1) We develop CALM, a contention-aware mechanism through co-designing the microarchitecture and runtime system of domain-specific accelerators (DSAs) to adaptively adjust the contentiousness of co-located applications in the presence of system-level contention.
2) We present two adaptation policies to handle different types of system-level contention: `balanced` policy when both co-located applications have priority, while `guaranteed` when a specific QoS target is only provided for the high-priority application.
3) We demonstrate the effectiveness of CALM on a multi-accelerator SoC using FPGA emulation. Our thorough evaluation shows CALM can increase the SLA satisfaction rate over state-of-the-art approaches while significantly decreasing the undesired performance variations.

II. Background and Motivation

A. Modern DNN Applications

With the wide adoption of DNN algorithms, modern applications include a large number of DNN workloads to support different concurrent functionalities. For example, robotics [6], [7] and AR/VR [3], [4], [27] applications need to detect nearby actors, track their movement, and predict action paths, all of which are simultaneous and DNN-based. These workloads differ greatly in latency and bandwidth requirements, ranging from real-time workloads with strict QoS deadlines, e.g., eye tracking in AR/VR devices [4], to offline workloads which are typically only run when the SoC is largely idle, e.g., identifying the people and objects in a photo album [9]. In these cases, multi-tenancy execution is a natural solution to avoid over -provisioning hardware resources.

B. Challenges with Multi-Tenancy Execution

While multi-tenancy execution generally improves hardware utilization, it also often faces performance degradation or non-idealities due to shared resource contention induced interference. Even in the context of DSAs, while each individual accelerator is generally designed in isolation, once integrated into an SoC, they typically share a number of system resources, including last-level cache (LLC), system bus, DRAM, and other I/O devices. As a result, co-located applications running on the accelerators can also place varying amounts of demand on these shared resources, leading to significant performance degradation due to system-level contention.

Figure 1 shows the performance degradation caused by co-locating two different DNNs together on a state-of-the-art SoC with DNN accelerators, general-purpose cores, shared system memory, and DRAM, with configurations similar to modern SoCs like NVIDIA's Xavier [28]. The DNN accelerator has an array of processing elements (PEs) and local scratchpads, partitioned equally between the co-located applications. We run each pair of co-located applications over 50 runs with randomized starting points to capture different interference patterns and measure their execution latency using FireSim, an FPGA-based, cycle-accurate RTL simulator [29]. Performance is normalized to the isolated execution latency where each application runs without any system-level interference.

We highlight two major challenges of multi-tenancy execution. The first one is the undesired performance degradation from system-level interference. We see that only one pair (AlexNet + GoogleNet) is able to meet the 90% QoS threshold, while all others experience significant performance degradation for at least one or both co-located applications. At the same time, one application could overwhelmingly starve its co-runners, as in the case of ResNet50, where it achieves more than 95% of its isolated performance but seriously degrades co-located applications by more than 30%.

The second challenge is the significant performance variation observed across runs. Many of today's DNN applications have strict SLA requirements across different scenarios, e.g., object detection in self-driving cars and personalized recommendation system in data centers [9]. We observe high performance variation especially when a large network like ResNet50 with high compute and memory requirements co-locates with a small network like SqueezeNet. Depending on which subset layers of large networks are co-located with small networks, there could be significant performance variation for the small ones due to the different demands in shared system-level resources. Both the latency degradation and high performance variation caused by system-level contention make it challenging to simply partition the accelerator resources to support multi-tenancy execution.

C. Architectural Support for Multi-Tenancy

More recently, driven by the strong demand to support multi-tenancy execution in DSAs [31], novel architectures have been proposed to support multi-tenancy execution of DNN accelerators through either temporal or spatial partition of on-chip resources, shown in Table I. While these techniques are effective

	Temporal Partition	Spatial Partition	
		Local Resource	System Resource
Static	LayerWeaver [22]	HDA [5]	CachePartition [11] Arbitration [30]
Dynamic	AI-MT [23] Prema [24]	Planaria [25]	**CALM**

TABLE I: Multi-tenancy support in DNN accelerators.

in running multiple applications on standalone accelerators, they have been mostly focusing on partitioning accelerator local resources, such as PEs and scratchpads, without considering the system-level resource contention at shared system memory, interconnect, and DRAM bandwidth. As a result, they cannot adaptively manipulate the accelerator's execution based on the contentiousness of current SoC execution, leading to sub-optimal performance for co-located applications. To the best of our knowledge, CALM is the first work that attacks the system-level resource sharing problem in DNN accelerators through co-designing the microarchitecture and runtime system of DNN accelerators to dynamically sustain QoS in the presence of system-level interference.

III. CALM ARCHITECTURE

To reduce the system-level performance interference of multi-tenant DNN accelerators, we propose CALM, a contention-aware DNN accelerator architecture that dynamically adjusts the contentiousness of co-located DNN workloads to shared system resources. CALM consists of 1) a light-weight hardware monitoring and regulation engine to track and control the amount of memory traffic injected into the shared resources and 2) an intelligent runtime system to dynamically manage the contentiousness of workloads based on user-specified adaptation policies. In particular, CALM leverages the regularity of DNN operators and hardware to manipulate the memory access rates of co-located workloads based on their latency targets, using two adaptation policies: `balanced` and `guaranteed` policy.

In this section, we first provide a high-level overview of CALM. Next, we describe the hardware and software components in detail, including the micro-architecture that enables memory traffic monitoring and throttling, and the runtime system that dynamically tunes the co-located performance by parameter configuration for the hardware engine.

A. CALM *Overview*

CALM is composed of a hardware component that monitors and limits the memory accesses of co-located DNN workloads and a runtime system that modulates the processing rate of each workload based on the system contentiousness and current adaptation policy, as shown in Figure 2. Leveraging the predictable computational and memory access patterns of DNN workloads, CALM runtime can accurately estimate the processing rate based on the performance targets for each layer and update them during the execution. If a workload currently falls short of its performance target, CALM runtime instructs CALM hardware to increase its target for concurrent

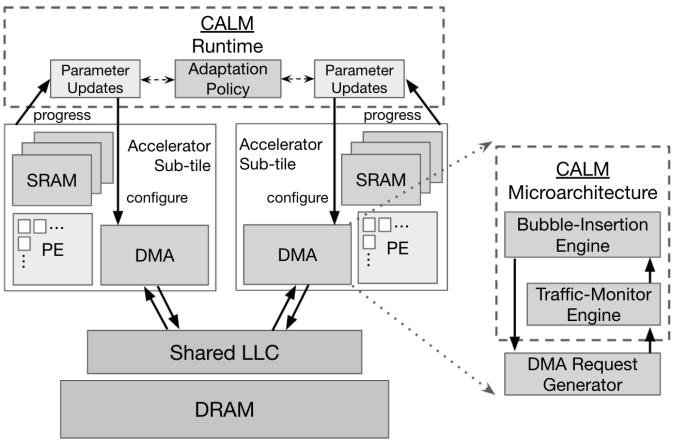


Fig. 2: CALM-augmented DNN accelerators on an SoC to support dynamic contention-aware execution.

memory accesses to boost its performance. Alternatively, if a workload runs ahead, CALM runtime temporarily slows it down to avoid over-provisioning of the shared resource and improve overall system throughput. Together, the hardware and software components of CALM reduce system-level contention of co-location and deliver predictable performance across runs.

CALM includes two adaptation policies to guide its dynamic contention modulation: the `balanced` mode, where all co-located applications have the same priority, and the `guaranteed` mode, where one latency-critical workload is co-located with low-priority workloads for shared resources.

In `balanced` mode, each DNN workload has own performance target, where CALM microarchitecture adjusts the memory traffic of each layer based on the target set by CALM runtime. Since the degree of interference each accelerator sees varies during the end-to-end run, CALM runtime monitors each workload's progress and adaptively adjusts its performance and memory traffic in real-time.

CALM's `guaranteed` mode aims at scenarios where extremely latency-critical workloads are co-located with other workloads that do not have explicit targets to meet, which we refer to as "offline" workloads [9]. Whenever layers of the latency-critical workload fall behind their performance targets, CALM runtime notifies the CALM microarchitecture to reduce the request rate of offline workloads made to shared caches and DRAM. At the same time, the `guaranteed` mode avoids the latency-critical application from over-claiming more shared resources than necessary, so that it does not throttle the co-located offline applications beyond what is necessary to meet QoS targets, improving the overall system performance.

B. CALM *microarchitecture*

CALM microarchitecture resides in DNN accelerator DMA, monitoring and restricting access to shared memory resources. CALM microarchitecture consists of two components as shown in Figure 3: `traffic-monitor engine` to track the real-time memory access rate at which DMA issues load request to shared caches and DRAM during the monitored time window,

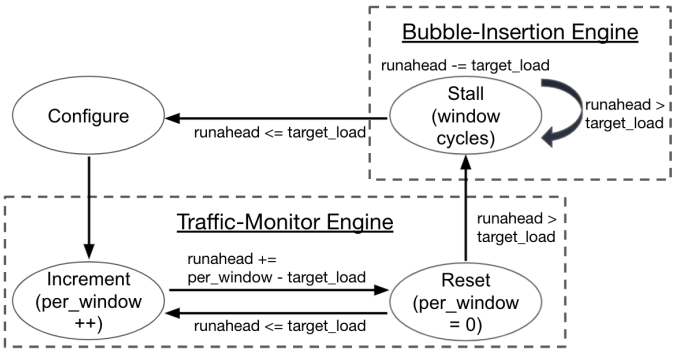


Fig. 3: CALM microarchitecture consists of a Traffic-Monitor Engine to track *Granted* load requests, and a Bubble-Insertion Engine that prevents DMA from requesting further load when it has reached the target number of load requests.

and `bubble-insertion engine` that inserts "bubbles", i.e., prevents the DMA from sending any further memory requests. CALM microarchitecture takes two parameters: *window* and *target load*, which are the time frame window memory traffic is monitored, and the target number of load requests we would like to issue to caches and DRAM during this time frame, respectively[1]. These parameters are calculated by CALM runtime based on memory traffic that is required to achieve the target latency for each layer. These hardware units are implemented mainly as light-weight finite state machines and counters, thus having quite small area overhead.

a) Traffic-Monitor Engine: The traffic-monitor engine monitors accelerators' DMA requests to check whether the current DNN layers are receiving enough traffic to achieve their performance targets, or whether DMAs from different accelerators are contending over shared memory resources. Depending on the SoC bus protocols, the traffic-monitor engine may evaluate this contention using different techniques. In our RISC-V-based implementation, we use the TileLink protocol to connect CPU cores, accelerators, and shared memory resources together [32]. When an accelerator sends a DMA request through the TileLink interface, the request can either be "granted" if it can be served immediately or "rejected" if the shared memory cannot accept more requests as it is already busy serving other requests. As Figure 3 (bottom) shows, the traffic-monitor engine counts the number of *granted* signals received during the *window*, i.e., the `per_window` counter. Whenever the number of *granted* requests exceeds the *target load*, the traffic monitor increments the delta to the `runahead` counter. Once the `runahead` counter reaches to the *target load*, i.e., the current execution has run ahead of its targeted run by a full *window*, it alerts the bubble-insertion engine to begin stalling DMA load requests. While our implementation is TileLink specific, the granted/rejected response can also be found in other SoC protocols like the AXI interface.

[1]We only tract the number of load requests as the DNN layer performance is more sensitive to the load request rates while the number of store requests are typically smaller and can be well overlapped by the execution phase.

b) Bubble-Insertion Engine: Based on the rate at which the DMA issues load requests (as monitored by the traffic-monitor engine), the bubble-insertion engine determines how many cycles "bubbles" should be inserted for, i.e., blocking further DMA requests. *Bubbles* prevent the DMA from making further load requests, which allows CALM to minimize memory contention while providing each accelerator with just enough memory bandwidth to achieve the target performance of whichever DNN layer it is running.

As shown in Figure 3, whenever the traffic-monitor engine raises an alert that its accumulated runahead counter exceeds *target load*, CALM microarchitecture transitions to "Stall" state, where the bubble-insertion engine begins stalling for *window* cycles. These two parameters are configured by CALM runtime, described further in Section III-C.

Every time bubble-insertion engine finishes injecting *bubbles*, it checks whether DMA's configuration register has been updated by CALM runtime, which happen when moving from one DNN layer to another (or one tile to another within a layer), that may have quite different tensor shapes and arithmetic intensities. After checking the configuration registers, CALM microarchitecture transitions back to Monitor state and begins incrementing the Inner counters based on memory traffic again.

C. CALM *runtime*

CALM runtime is responsible for calculating performance and utilization targets for each DNN layer based on the QoS targets, setting the memory access rate that is required to achieve that target utilization, and configuring the hardware based on these calculations. The tensor shapes of DNN workloads are statically known ahead-of-time, allowing CALM runtime to accurately calculate how much memory bandwidth should be shared to allow each workload to fulfill its performance target. In addition, CALM runtime updates these performance targets dynamically during execution as the executed layers may run ahead or fall behind the target latency, which could impact the required latency for the layers that have not been executed.

Specifically, to accurately set the latency target of each DNN layer, CALM runtime categorizes DNN layers into two types: `compute` layers which are compute-bound (such as convolutional layers), and `mem` layers which are memory-bandwidth-bound (such as fully-connected layers and residual additions). Before execution, CALM runtime first maps each layer to one of these types and estimates the execution latency for all `mem` layers, based on the shared memory bandwidth available on the SoC. CALM runtime then sets a target runtime for `compute` layers by subtracting the latency of `mem` layers from the target latency for the entire DNN workload that is derived from its QoS requirement. CALM runtime sets the performance targets of `mem` layers first, because they are far more likely to degrade performance by contending over shared memory resources, and therefore determine the degree to which the `compute` layers must make up for this contention by surrendering access to the cache or DRAM hierarchy. Next, we first discuss how CALM runtime sets the initial latency estimation for each individual layer before execution, followed

by how it dynamically configures CALM microarchitecture and updates the layer latency targets based on runtime information.

1) Initial Latency Estimation: Algorithm 1 details how CALM runtime initially sets the layer execution target.

Algorithm 1 CALM runtime initial latency estimation

1: total_runtime ← getTargetRuntime(QoS_target)
2: total_memtime = 0
3: **for** i in FC layers **do**
4: $FC\_time \leftarrow \alpha \cdot \frac{In\_A_i + In\_B_i}{BW_{Accel}}$
5: $total\_memtime+ = FC\_time$
6: **end for**
7: **for** i in residual-addition layers **do**
8: $resadd\_time \leftarrow \alpha \cdot \frac{In\_A_i + In\_B_i + Out_i}{BW_{Accel}}$
9: $total\_memtime+ = resadd\_time$
10: **end for**
11: **for** i in pooling layers **do**
12: $pool\_time \leftarrow \alpha \cdot \frac{In_i + Out_i}{BW_{Accel}} + \frac{OUT_i \cdot Window^2}{pool\_throughput}$
13: $total\_memtime+ = pool\_time$
14: **end for**
15: $total\_comptime = total\_runtime - total\_memtime$
16: $total\_comp\_ops \leftarrow$ getTotalCompOps()
17: $comp\_ideal\_runtime \leftarrow total\_comp\_ops / num\_PE$
18: $comp\_target\_util\_initial \leftarrow \frac{comp\_ideal\_runtime}{total\_comptime}$

a) `mem` layers: As described in lines 2 to 17 of Algorithm 1, CALM runtime calculates performance targets for each `mem` layer by iterating over all memory-bound layers, e.g., fully-connected layers and residual additions, and calculates all their expected runtimes based on the operand sizes and available memory bandwidth. The total expected runtime of `mem` layers will be used later to set performance targets for compute-bound layers. In Algorithm 1, BW_{Accel} represents the total memory bandwidth on the SoC. α is a scalar term applied to BW_{Accel} to account for scenarios where the full memory bandwidth can not realistically be achieved. For example, if four different accelerators on an SoC are sharing the same DRAM, then α may be set to four, since each accelerator would be expected to utilize one-fourth of that total bandwidth.

b) `compute` layers: After the performance targets of the memory-bound layers have been set, CALM runtime calculates the target `compute` layer runtime by subtracting the total `mem` runtime from the overall target runtime. CALM runtime then calculates the target resource utilization each compute-bound layer requires to achieve the overall target performance. Line 19 to 22 in Algorithm 1 illustrate the process.

2) Configuring CALM *hardware:* During runtime, CALM runtime sets the hardware parameters, `window` and `target_load`, such that the accelerator can make just enough load requests to meet the required target runtime for current layer. Each DNN layer's target runtime is calculated based on the number of memory or arithmetic operations that it must perform and the target utilization that it must achieve to maintain QoS requirements. For `mem` layers, target runtime would depend on shared memory bandwidth and bandwidth

Algorithm 2 Parameter configuration and runtime updates

1: $remain\_runtime = total\_runtime$
2: $remain\_memtime = total\_memtime$
3: $remain\_ops = total\_comps\_ops$
▷ % Running each layer of the network during runtime %
4: **for** i in total layers **do**
▷ % Calculating `window` and `target_load` %
5: $target\_util \leftarrow$ i is `mem`? $\frac{1}{\alpha}$: comp_target_util
6: $ops\_per\_cycle \leftarrow$ i is `mem`? BW_{Accel} : num_PE
7: $curr\_ops \leftarrow$ getLayerOps(i)
8: $curr\_tile \leftarrow$ getLayerTile(i)
9: $layer\_target\_runtime \leftarrow \frac{curr\_ops}{ops\_per\_cycle \cdot target\_util}$
10: `window` $\leftarrow layer\_target\_runtime / curr\_tile$
11: `target_load` $\leftarrow num\_load / curr\_tile$
12:
▷ % Running the current layer %
13: runLayer(i, `window, target_Load`)
14: $curr\_runtime \leftarrow$ getLayerRealRuntime(i)
15:
▷ % Updating latency targets of following layers %
16: $remain\_runtime- = curr\_runtime$
17: **if** i is `mem` layer **then**
18: $remain\_memtime- =$ getMEMPrediction(i)
19: **end if**
20: **if** i is `compute` layer **then**
21: $remain\_ops- = curr\_ops$
22: **end if**
23: $comp\_runtime = remain\_runtime-$
24: $remain\_memtime$
25: $comp\_ideal\_runtime \leftarrow \frac{remain\_ops}{num\_PE}$
26: $comp\_target\_util \leftarrow \frac{comp\_ideal\_runtime}{comp\_runtime}$
27: **end for**

utilization while for `compute` layers, target runtime depends on the total number of available functional units which can perform arithmetic operations and target utilization of these functional units (which is called $compute\_target\_util$ in Algorithms 1 and 2). The `window` size is set to the expected duration of one iteration of a tile in a tiled implementation of a DNN kernel, where the tile size is determined to maximize the reuse of its available local scratchpad memory. The `target_load` is set to the number of load requests that will be generated per tile during `window` time frame.

3) Runtime updates: Memory contention over shared resources is quite difficult to predict before DNN workloads run, due to the uncertainty which layers across different DNNs will be co-located at the same time on the same SoC. Ideally, we would like to co-locate memory-bound layers with compute-bound layers, but this is not always possible due to the unpredictability of co-located workloads. To account for this unpredictability and randomness, CALM iteratively adapts the target runtimes of DNN workloads dynamically, accounting for the unpredictable execution scenario that causes individual layers to take longer or shorter than expected, as shown in

Algorithm 3 CALM `guaranteed` adaptation policy

for i in total layers **do**
▷ % After the updates from Algorithm 2, the adaptation policy further adjusts the target utilization based on the priority of co-located applications. %
  if `guaranteed` && app is *high_priority* **then**
    $gap = comp\_target\_util -$
      $comp\_target\_util\_initial$
  end if
  if `guaranteed` && app is *low_priority* **then**
  ▷ % Adjust the target based on the progress of its high-priority co-runners %
    $comp\_target\_util \times = \frac{100}{100+gap}$
  end if
end for

Algorithm 2. Whenever `compute` layers exceed their runtime targets, the target utilization for future layers will be increased, so that the runtime of the overall DNN workload remains within the boundaries set by QoS requirements. Similarly, if a layer runs ahead of its target, the target utilization for future layers will be dropped accordingly, so that it won't overclaim unnecessary resources from its co-runners.

D. CALM *Adaptation Policies*

CALM includes two adaptation policies, `guaranteed` and `balanced`, to cope with the diverse performance requirements across different co-location scenarios. The `guaranteed` policy prioritizes a single DNN latency-critical workload while co-locating with offline applications, while `balanced` attempts to balance memory access across multiple workloads, each with own QoS requirements.

a) `guaranteed`: When a latency-critical workload with strict SLA requirements is co-located with offline workloads without strict latency targets, the `guaranteed` policy attempts to guarantee that we meet the QoS requirements of the latency-critical workload, while still maximizing the throughput of low-priority workloads. Algorithm 3 illustrates how the `guaranteed` dynamically throttles the memory accesses of offline applications whenever they are threatening to interfere with the memory accesses of the latency-critical application.

Whenever layers of the latency-critical application perform slower or faster than expected, CALM runtime updates the global `gap` parameter, which measures the `gap` between the adjusted performance and the initial performance goals of the high-priority application. If `gap` is positive, i.e., the currently required performance is higher than the initial target, meaning the high-priority application is running behind the performance goal. As a result, CALM runtime will slow down the low-priority application accordingly so that the high-priority application can catch up. Using this mechanism, latency-critical applications can carefully slow down the memory request rates of co-located workload, reducing their memory interference level and successfully boosting their own performance.

Parameter	Value
Systolic array dimension (per subtile)	16x16
Inner scratchpad size (per subtile)	64KiB
Inner accumulator size (per subtile)	32KiB
# of accelerator subtiles	4
Shared L2 size	2MB
Shared L2 banks	4
DRAM bandwidth	16GB/s
Frequency	1GHz

TABLE II: SoC configurations used in evaluation.

b) `balanced`: In addition, CALM runtime also supports a `balanced` policy that is designed to equally support co-located applications and meet their individual QoS targets, where these QoS requirements could all be achieved if SoC resources are partitioned by a perfect co-location scheme. In `balanced`, the CALM runtime calculates the target performance of each DNN layer in each workload and the required memory bandwidth each layer needs to achieve that performance. The CALM microarchitecture monitors and controls the DMA usage so that each workload can achieve its target utilization as closely as possible without over-utilizing shared memory resources, avoiding hurting the performance of other workloads. This helps the overall system to fairly and dynamically partition the SoC shared resources and decreases the unpredictable variance across multiple runs by enforcing the target performance across runs. CALM's `balanced` can also increase overall system throughput by preventing cache thrashing and other effects of memory over-utilization.

IV. Methodology

A. CALM *implementation*

We implement the CALM microarchitecture using the Chisel RTL language [33] on top of the Gemmini [34] infrastructure, a systolic-array-based DNN accelerator without multi-tenancy support. We evaluate performance on full, end-to-end runs of DNN workloads using FireSim, a cycle-accurate, FPGA-accelerated RTL simulator [29]. We also synthesize our hardware implementations on the Global Foundry 12nm process to evaluate the area overhead.

Table II shows the SoC configuration we use in our evaluations of CALM, similar to the configurations in modern SoCs [28]. We configure Gemmini to produce four separate accelerator subtiles on the same SoC, each of which can run a *different* DNN workload. Multiple accelerator subtiles can also cooperate together to run different layers/regions of the *same* DNN workload. Each accelerator subtile is equipped with a 16x16 systolic array for matrix multiplications and convolutions, as well as a private scratchpad memory. All subtiles also share access to the shared memory subsystem, including a shared last-level cache and DRAM, which is the main source of contention with co-located workloads.

B. Workloads

In our evaluations, we process two different DNN workloads simultaneously by spatially co-locating them using CALM. We

Model size	Workloads	Optimal QoS	Optimal FPS
Small	SqueezeNet	1.72 ms	581
Mid	GoogLeNet	7.17 ms	139
Large	AlexNet	10.98 ms	91
	ResNet50	13.99 ms	71

TABLE III: Optimal QoS for evaluated workloads measured when they run in isolation w/o system-level interference.

evaluate two different co-location scenarios: (i) a latency-critical workload is co-located with a low-priority offline workload, and (ii) two different DNN workloads of the same priority are co-located, each of which has its own QoS target. The first scenario allows us to evaluate whether CALM can meet the QoS targets of latency-critical applications, while still maintaining (or even improving) full system throughput for offline applications.

We evaluate a diverse set of state-of-the-art DNN inference models, including SqueezeNet [35], GoogLeNet [36], AlexNet [37], and ResNet [1]. Each model runs with a batch size of one, and we run each workload with 50-150 different starting points to cover different co-locating scenarios where different layers are overlapped with each other. These DNN models represent a diverse set of DNN workloads, with different model sizes, DNN kernel types, computational requirements, and compute-vs-memory trade-offs.

For example, SqueezeNet is a small model with a short runtime and relatively small layer dimensions, while GoogLeNet is a mid-sized model that concatenates the outputs of multiple different layers together, increasing pressure on shared caches to store intermediate outputs that will not be reused until several other operations are completed. AlexNet and ResNet50 are both large networks with different characteristics. AlexNet is a traditional network with multiple FC layers that are highly memory-bounded with large weight sizes, while ResNet50 has large channel sizes in its CONV layers, especially towards the end of the network, increasing the risk of thrashing on shared caches. In addition, the residual addition layers are highly memory-bounded, where inputs to the residual layers may be evicted when many workloads share the memory subsystem.

Table III shows the optimal QoS performance for our DNN workloads, which we set to the optimal performance of a standalone workload running on two accelerator subtiles without any system-level interference. We call this optimal QoS for multi-tenancy execution, MT_{opt}, to represent the upper bound of the performance that can be achieved when the application shares its memory resources with other co-located workloads, running on the other two accelerator subtiles of the SoC. Depending on the execution scenarios, deployment typically aims to achieve 85% to 95% of this optimal QoS performance with multi-tenancy execution [9]. We use the same range in our evaluation.

C. Metrics

We quantify the effectiveness of CALM-enabled workload co-location using the metrics suggested in [38], including the percentage of workloads for which we satisfy SLA, throughput of the co-located applications, and fairness of CALM's strategy for arbitrating access to shared memory resources. We also included the variance between runs as another metric to evaluate whether CALM is able to provide reliable performance even in the presence of unpredictable resource contention.

a) SLA satisfaction rate: We set the SLA target of each workload as a percentage of its optimal QoS performance, sweeping this target from 83% to 97%.

b) Fairness: We evaluate CALM's `balanced` policy using *fairness*, which measures the extent to which all programs have equal progress. Here, we use C_i to denote the cycle time of the i-th workload, $single$ represents only one workload running on the SoC, and MT represents the multi-tenant execution. Similar to other prior work in DNN multi-tenancy support [24], [25], we use a generalized version of *fairness* that measures *proportional progress* (PP) defined as follows:

$$PP_i = \frac{\frac{C_i^{single}}{C_i^{MT}}}{\frac{Priority_i}{\sum_{j=1}^{n} Priority_j}} \qquad Fairness = min_{i,j} \frac{PP_i}{PP_j} \tag{1}$$

c) Throughput: We also analyze the total *system throughput* (STP), where the system throughput of executing n programs is defined by summing each program's normalized progress ranging from 1 to n. Increasing STP requires maximizing overall progress when co-locating multiple applications.

$$STP = \sum_{i=1}^{n} \frac{C_i^{MT}}{C_i^{single}} \tag{2}$$

d) Variance: Finally, we measure the performance variation under contention using the *coefficient of variation* (CoV). This metric is the standard deviation of each application's performance over the average, expressed as follows:

$$CoV_i = \frac{\sigma_i^{MT}}{C_i^{MT}} \tag{3}$$

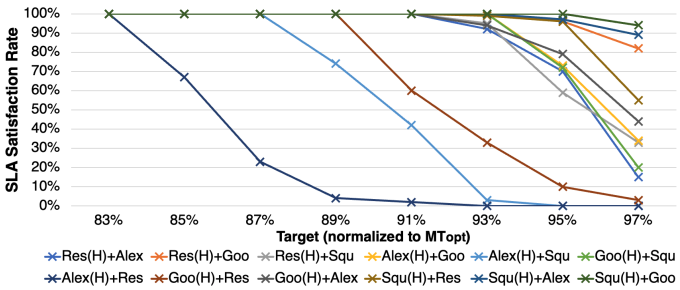
where σ_{MT} is the standard deviation of the multi-tenancy execution performance. By analyzing the CoV, we can evaluate whether CALM is able to generate deterministic and consistent performance across different runtime scenarios.

D. Baseline

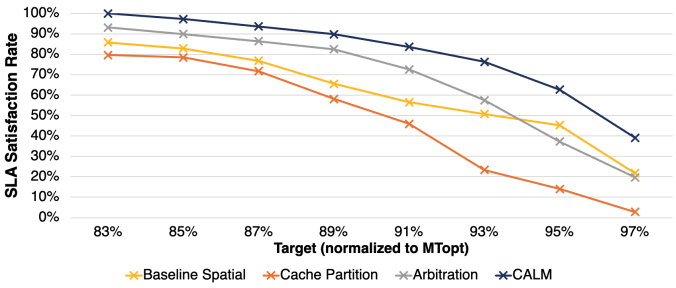
We compare CALM with four state-of-the-art co-location mechanisms: `temporal` co-location where workloads are interleaved temporally during execution; `spatial` co-location where workloads are running simultaneously with resource shared spatially; `cache partition` where the shared LLC is partitioned statically across workloads; and `priority arbitration` where we statically prioritize memory requests from high-priority workloads in system-level interconnect.

For cache partitioning, in the `guaranteed` mode, we partition an 8-way cache statically, by allocating six ways to the high-priority workload and the remaining two to the low-priority workload. For the `balanced` mode, on the other hand, we partition it equally, 4 ways each, to improve fairness and throughput by reducing shared memory space interference.

For priority arbitration, we modify the LLC arbiter to have input queues with different priority levels for each cache bank and statically prioritize queued requests [39], [40] from the



(a) CALM's SLA satisfaction rate across different workload pairs.



(b) CALM's SLA satisfaction rate over other spatial co-locations.

Fig. 4: SLA satisfaction rate of the latency-critical workload with CALM's `guaranteed` policy with different performance targets. (H) denotes the high-priority application. CALM significantly improves the SLA satisfaction rates.

latency-critical application. This allows statically prioritizing the shared cache access of the latency-critical workloads if there are concurrent memory requests from different accelerator subtiles. In our implementation, we design the priority arbitration scheme on top of the Tilelink protocol [32] but it can also be applied to other shared resource arbitration schemes.

V. Evaluation

In this section, we evaluate the effectiveness of CALM-enabled workload co-location in meeting the target QoS of a variety of DNN applications using the metrics and baseline comparisons described in Section IV. We evaluate different workload scenarios, including scenarios where a single latency-critical workload is co-located with offline applications and where multiple applications with different QoS targets compete for shared resources. Finally, we demonstrate that CALM's hardware components imposes a small area overhead.

A. CALM `guaranteed` mode effectiveness

As Figure 4a shows, we tested twelve combinations of latency-critical, denoted with H, and offline DNN workloads, sweeping our SLA requirements from 83%, up to an aggressive 97% of the optimal un-interfered performance. We observe that even with very aggressive QoS requirements, CALM enables latency-critical applications to preserve 95% of their optimal performance on over 80% of runs.

As shown in Figure 4b, CALM also outperforms all our static baseline mechanisms, ranging from straightforward partitions to more sophisticated cache partitioning and arbitration strategies as described in Section IV. We increase the SLA satisfaction rate by 8.6× over the naive co-location strategy at most, and by 2.3× overall. CALM achieves higher SLA satisfaction rates than our baseline solutions for all the QoS requirements that we swept, even for the most lenient ones, where memory contention is not as strong of a limiting factor. Figure 5 further illustrates that our improvement over other partitioning strategies holds for a variety of different model shapes and sizes, including both small models like SqueezeNet, and much larger models like AlexNet, which is the largest model that we evaluate.

Static cache partition is the least effective strategy we evaluate, performing worse than baseline spatial co-location, which demonstrates inflexible static partitioning of memory capacities

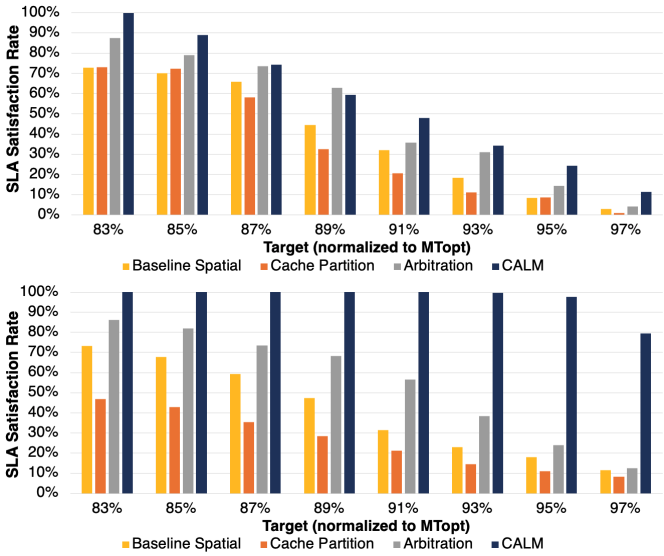


Fig. 5: SLA satisfaction rate breakdown for AlexNet (top) and SqueezeNet (bottom) across different performance targets.

can worsen the performance since memory requirements can change dramatically over the end-to-end run of co-located workloads. The reason is that with a static partition strategy, co-located workloads either fail to utilize their partition effectively or cannot fit all their layer in/outputs inside of it, leading to excessive cache misses. The increase in cache misses also causes cache evictions to compete with other DNN layers that are attempting to load weights from DRAM. These interactions between cache sizes, cache access strategies, and DRAM bandwidth utilization can be difficult to predict statically, but CALM's dynamic monitoring solution is able to accommodate them to achieve much higher SLA satisfaction rates.

We also notice that our baseline arbitration strategy, which always prioritizes requests from the latency-critical application, also does worse at satisfying SLA requirements of the latency-critical application than CALM, as illustrated in Figure 4b. Although an intuitive impulse is to always prioritize the requests of latency-critical applications, we find that the arbitration strategy actually causes more memory contention than CALM, which hurts the memory access latency of the latency-

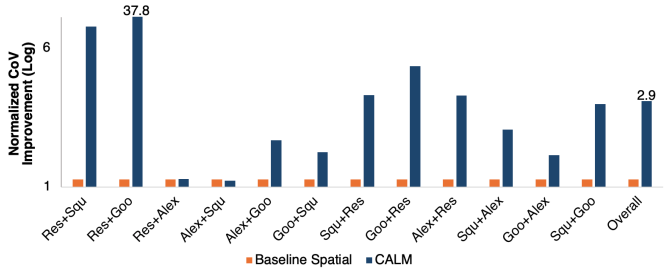


Fig. 6: Performance variation improvement of CALM compared to the spatial baseline with an SLA target of 95%.

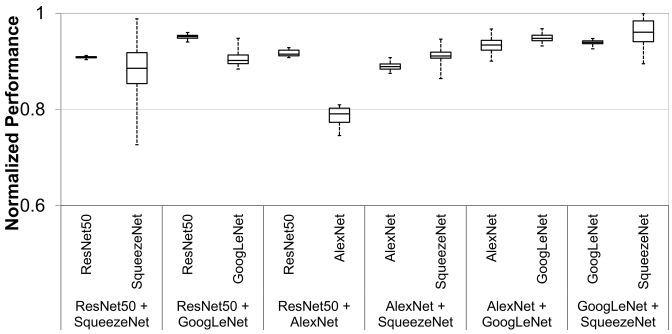


Fig. 7: Normalized performance distribution of different co-locating workloads with CALM's `balanced` mode.

critical workload. Instead, CALM leverages model-specific information to enforce the performance of both high- and low-priority workloads to only what is necessary to achieve SLA requirements, which reduces the negative effects of memory contention. CALM thus exploits domain-specific knowledge of DNN layers to minimize shared resource contention, achieving significantly higher performance than the baselines.

CALM struggles most to achieve SLA requirements when co-locating two large, memory-bound workloads, e.g., when AlexNet and ResNet50 are co-located. AlexNet and ResNet50 both spend a large portion of their runtime on memory-bound layers, like AlexNet's FC layers or ResNet50's later CONV layers with relatively small weight reuse. Co-locating these memory-bound layers is intrinsically challenging as they reduce the opportunity to overlap the memory-bounded layers of one network with the compute-bounded layers of another network.

In addition, CALM also significantly improves the consistency of DNN workload performance, even when co-location causes unpredictable SoC resource contention. Figure 6 shows the performance variance improvement of the latency-critical applications of CALM-enabled co-location over the baseline spatial co-location, measured using the CoV metric defined in Equation 3. We see that CALM reduces the variance by a factor of 2.9× on average across different co-locating pairs compared to the baseline spatial co-location strategy.

B. CALM `balanced` *mode effectiveness*

When co-located workloads have the same priority, CALM operates in the `balanced` mode, which attempts to meet the target QoS of each DNN workloads simultaneously. Different from the `guaranteed` mode, `balanced` does not assume that one of the workloads is more critical than others. Instead, it attempts to establish a balanced execution environment where every workload gets to achieve its performance target. To evaluate the performance of the `balanced` mode, we set an ambitious target QoS for *all* our DNN workloads to achieve 90% of their optimal un-interfered performance.

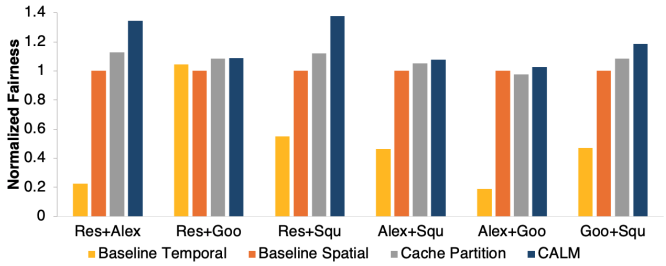
Figure 7 shows the resulting normalized performance distribution of the co-located DNN models with CALM's `balanced` mode. Compared to the baseline spatial co-location strategy in Figure 1, we observe that not only the performance of both co-located pairs increase but also their performance distributions are much more consistent, i.e., significantly improving the performance variations across runs. This effect is more pronounced when there is a large imbalance between the model sizes of the co-located workloads. Baseline spatial co-location strategy cannot prevent the larger models from monopolizing the majority of shared memory capacity and DRAM bandwidth, significantly hurting performance and SLA satisfaction rate for smaller, co-located models. The one exception is when AlexNet is co-located with ResNet50, where the DRAM bandwidth requirements of AlexNet's FC layers and ResNet50's later layers make it infeasible to achieve such aggressive performance targets.

In particular, to reduce memory contention, CALM prevents the large workloads from over-utilizing memory resources above needed to achieve their target latencies. This causes a small decrease in the performance of these larger models (but without violating their QoS requirements), compared to the baseline spatial co-location strategy. At the same time, it improves the performance of the smaller model which previously fell far short of their own QoS targets. For example, when a small model like SqueezeNet is co-located with a larger model like ResNet50, CALM's `balanced` mode slightly reduces the performance of the larger model by 7.5% (which is still within its QoS target of 90% of MT_{opt}) while increasing the performance of the smaller model by 16%.

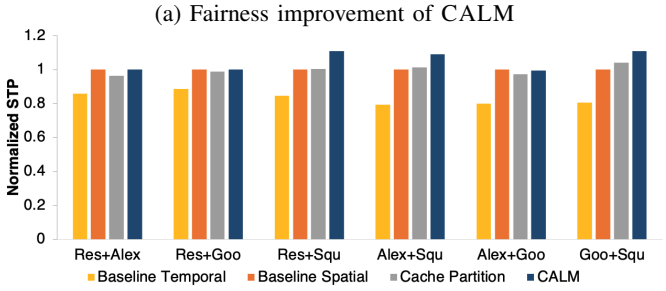
We notice that performance variation still exists for workload pairs like ResNet50 + SqueezeNet. This is because SqueezeNet has a far shorter runtime than ResNet50 so that an entire end-to-end run of SqueezeNet can complete in the time that ResNet50 completes only a couple of DNN layers. Depending on which subset of ResNet50 layers the SqueezeNet run overlaps with, memory contention can vary significantly. However, we show that the performance variation of SqueezeNet improves over 2.2× under CALM, discussed later in Figure 9.

In addition, we evaluate the effectiveness of `balanced` mode at maintaining fair access to shared memory, increasing overall throughput, and reducing performance variations.

a) Fairness: As illustrated in Figure 8a, CALM's `balanced` mode achieves better fairness scores (as described in Section IV) than all the other baseline co-location strategies that we test. The fairness improvement compared to the baseline spatial co-location strategy is greater for workload pairs involving larger DNN models. When other models are co-



(a) Fairness improvement of CALM



(b) STP improvement of CALM

Fig. 8: Fairness and STP improvement over evaluated multi-tenancy baselines (normalized to baseline spatial partition).

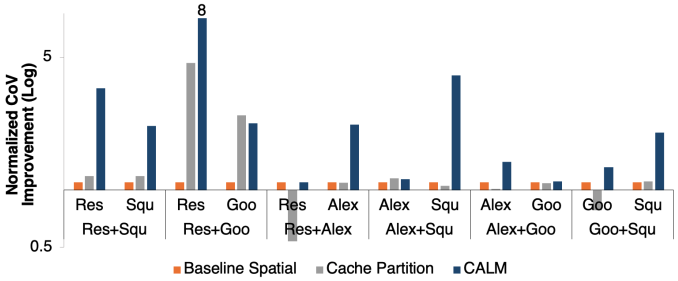


Fig. 9: Performance variation improvement of CALM over spatial co-location baselines.

located with ResNet50, the fairness degradation of the baseline strategies is quite pronounced because ResNet50 consumes a large portion of the shared cache space and consumes much DRAM bandwidth for later layers due to large weight size, causing a significant amount of interference, but CALM prevents any one model from dominating SoC resource usage, leading to improved fairness and more balanced execution.

b) Throughput: Figure 8b shows the system throughput improvement that CALM achieves over the baseline multi-tenancy strategies, normalized to the baseline spatial partition. We see that in addition to improving the fairness of the system as we discussed earlier, CALM also increases overall system throughput across different workload pairs. The increase is especially pronounced for small models like SqueezeNet, which due to their short runtimes, can suffer greater proportional reductions to their throughputs during periods of momentary unpredictable resource contention. By reducing the overall system-level contention, CALM's `balanced` mode allows the small co-located models to increase their throughput without significantly hurting other co-located models.

Component	Area w/o CALM (μm^2)	Area w/ CALM (μm^2)	Increase
Subtile area	472K	472K	0.02%
DMA area	9.2K	9.3K	1.7%

TABLE IV: Area comparison between default (w/o CALM) and proposed architecture (w/ CALM).

c) Variation evaluation: We also evaluate the performance consistency of the co-located DNN workloads by measuring CoV, defined in Equation 3. We run the DNN workloads with randomized starting points to maximize the combinations of different overlapping layers across different models, leading to varied system resource contention. As shown in Figure 9, CALM allowed the co-located workloads to achieve much more consistent performance, on average, than both the baseline spatial co-location scheme and the cache partitioning scheme.

C. Area analysis

As shown in Table IV, CALM adds a minimal area overhead to DNN accelerators, increasing the size of DMA by only 1.7%, and the area of entire accelerator (including its functional units) by less than 1%, because DMA itself is a small component of the accelerator. This demonstrates that CALM's performance improvements can be obtained with minimal area overhead. We synthesize both CALM and the accelerator using Cadence Genus with Global Foundry 12nm process technology.

VI. Conclusion

We propose CALM, a contention-aware, multi-tenancy accelerator architecture for DNN workloads. CALM leverages the regularity of DNN layers and hardware to estimate the expected latency of each DNN layer to meet the QoS targets of co-located applications. Specifically, CALM dynamically monitors the performance of each layer at runtime, adaptively updates their performance targets on the real-time performance of executed layers, and throttles memory accesses if needed to avoid undesired system-level interference with two adaptation policies: `guaranteed` and `balanced`. Our thorough evaluation demonstrates that CALM increases SLA satisfaction rates up to 8.6× over baseline spatial multi-tenancy strategies, while reducing the variation of workload performance between runs by up to 37.8×. CALM also increases system throughput and fairness up to 1.11× and 1.38×, respectively, while incurring less than 1% area overhead of a state-of-the-art DNN accelerator.

We hope to explore the implementation of CALM in heterogeneous domain specific architectures system such as CALM integrated with vector accelerator [41] to characterize the behavior of our system. We expect there to be accelerator specific parameters and modifications to be done to CALM in order to adapt our system to other types of accelerators in the SoC. Finally, we also look forward to benchmarking CALM with diversified scenarios like co-locating image segmentation, path finding, keyword spotting, and natural language processing workloads to prove the significance in moving towards designing QoS-aware architecture with CALM.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, 2018.

[3] M. Huzaifa, R. Desai, X. Jiang, J. Ravichandran, F. Sinclair, and S. V. Adve, "Exploring extended reality with ILLIXR: A new playground for architecture research," *IISWC*, 2021.

[4] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine Learning at Facebook: Understanding Inference at the Edge," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[5] H. Kwon, L. Lai, T. Krishna, and V. Chandra, "HERALD: optimizing heterogeneous DNN accelerators for edge devices," *CoRR*, vol. abs/1909.07437, 2019.

[6] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. J. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.

[7] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, *et al.*, "The limits and potentials of deep learning for robotics," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 405–420, 2018.

[8] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. A. Patterson, H. Tang, G. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. M. Hazelwood, A. Hock, X. Huang, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, C. Wu, L. Xu, C. Young, and M. Zaharia, "Mlperf training benchmark," *CoRR*, vol. abs/1910.01500, 2019.

[9] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," *CoRR*, vol. abs/1911.02549, 2019.

[10] Q. Chen, H. Yang, M. Guo, R. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 17–32, 04 2017.

[11] H. Cook, M. Moreto, S. Bird, K. Dao, D. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," *ACM SIGARCH Computer Architecture News*, vol. 41, 06 2013.

[12] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013.

[13] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2014.

[14] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2012.

[15] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," vol. 41, no. 1, 2013.

[16] J. Mars, N. Vachharajani, R. Hundt, and M. L. Souffa, "Contention aware execution: Online contention detection and response," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2010.

[17] Q. C. 0002, 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*, pp. 681–696, ACM, 2016.

[18] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021.

[19] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[20] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi, "Missing the Forest for the Trees: End-to-End AI Application Performance in Edge Data Centers," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[21] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[22] 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 *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[23] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 940–953, 2020.

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

[25] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 681–697, 2020.

[26] M. D. Hill and V. J. Reddi, "Accelerator-level parallelism," *CoRR*, vol. abs/1907.02064, 2019.

[27] H. Kwon, L. Lai, M. Pellauer, Y.-H. Chen, T. Krishna, and V. Chandra, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[28] F. Sijstermans, "The NVIDIA Deep Learning Accelerator," in *Hot Chips*, 2018.

[29] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 29–42, 2018.

[30] S. Tiwari, S. Tuli, I. Ahmad, A. Agarwal, P. R. Panda, and S. Subramoney, "Real: Request arbitration in last level caches," *ACM Trans. Embed. Comput. Syst.*, 2019.

[31] "Facebook Architecture, Compiler, and System Support for Multi-Model DNN Workloads Workshop at MICRO." https://research.fb.com/arch-comp-sys-support-for-multi-model-dnn-workshop/, 2021.

[32] H. M. Cook, A. S. Waterman, and Y. Lee, "Sifive tilelink specification," tech. rep., SiFive Inc., 2018. https://www.sifive.com/documentation/tilelink/tilelink-spec/.

[33] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Design Automation Conference*, 2012.

[34] 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 *Design Automation Conference*, 2021.

[35] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[36] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (Red Hook, NY, USA), p. 1097–1105, Curran Associates Inc., 2012.

[38] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[39] Y. Qian, Z. Lu, and Q. Dou, "Qos scheduling for nocs: Strict priority queueing versus weighted round robin," in *2010 IEEE International Conference on Computer Design*, pp. 52–59, 2010.

[40] C. Li, R. Bettati, and W. Zhao, "Static priority scheduling for atm networks," in *Proceedings Real-Time Systems Symposium*, pp. 264–273, 1997.

[41] Y. Lee, *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. PhD thesis, EECS Department, University of California, Berkeley, May 2016.