ODIN: Overcoming Dynamic Interference in iNference pipelines

Pirah Noor Soomro[0000−0001−8654−3249], Nikela Papadopoulou[0000−0003−2141−5654], and Miquel Pericàs[0000−0002−7583−6609]

Chalmers University of Technology, Gothenburg, Sweden
{pirah,nikela,miquelp}@chalmers.se

Abstract. As an increasing number of businesses becomes powered by machine-learning, inference becomes a core operation, with a growing trend to be offered as a service. In this context, the inference task must meet certain service-level objectives (SLOs), such as high throughput and low latency. However, these targets can be compromised by interference caused by long- or short-lived co-located tasks. Prior works focus on the generic problem of co-scheduling to mitigate the effect of interference on the performance-critical task. In this work, we focus on inference pipelines and propose ODIN, a technique to mitigate the effect of interference on the performance of the inference task, based on the online scheduling of the pipeline stages. Our technique detects interference online and automatically re-balances the pipeline stages to mitigate the performance degradation of the inference task. We demonstrate that ODIN successfully mitigates the effect of interference, sustaining the latency and throughput of CNN inference, and outperforms the least-loaded scheduling (LLS), a common technique for interference mitigation. Additionally, it is effective in maintaining service-level objectives for inference, and it is scalable to large network models executing on multiple processing elements.

Keywords: CNN parallel pipelines · Online tuning · Design space exploration · Interference mitigation · Inference serving

1 Introduction

As machine learning becomes the backbone of the digital world, there is an increasing demand for predictions as a service. This has led to the advent of inference-serving systems [7, 19, 21, 24, 25]. These systems deploy pre-trained model pipelines, i.e. inference pipelines, on the cloud, serving inference queries to users and applications, often under strict quality-of-service (QoS) requirements for the response times and throughput of the queries [32], expressed as service level objectives (SLOs). However, due to the limited availability of resources of cloud systems, in combination with high demand, inference pipelines are often co-located with other workloads, either as part of the inference-serving system, which may opt to co-locate multiple inference pipelines [22, 31], or as part of common multi-tenancy practices of cloud providers [9,10] to increase utilization.

The resulting interference from the co-located workload can have devastating effects on inference performance, leading to violation of the SLOs.

The mitigation of the effect of interference from co-located workloads on the performance of a critical application has been studied extensively. Several scheduling techniques focus on the generic problem of workload colocation, trying to retain or guarantee the performance of one critical or high-priority workload under interference [4, 5, 9, 10], while more recent works focus on the problem of colocating inference pipelines specifically [17, 22, 25]. Most of these techniques perform extensive offline profiling and/or characterization of workloads and workload colocations, and build pre-trained machine-learning models or analytical models for each system, while a brief profiling phase may also be required to characterize a workload [9, 10]. These techniques proactively partition resources to the workloads to mitigate the effect of interference, but may reactively repartition resources or evict colocated workloads in response to changes in the observed performance or interference. Finally, some techniques only focus on interference effects affecting specific resources, such as GPU accelerators [4,5].

One way to achieve high throughput and low latency for inference pipelines is pipeline parallelism. Pipeline parallelism in the form of layer pipelining has been used extensively in training [12,14,20,23], and in inference [16,30], in combination with operator parallelism, as it is able to reduce data movement costs. To exploit pipelined parallelism, several techniques focus on finding near-optimal pipeline schedules online, using heuristics to tackle the large search space [3, 15, 28, 29]. The ability to rebalance pipeline stages online leaves ample room for the optimization of the execution of a pipeline under the presence of interference, where such a reactive technique can detect and mitigate performance degradation, by making better utilization of the existing resources.

In this work, we propose ODIN, an online solution that dynamically detects interference and adapts the execution of inference pipelines on a given set of processing elements. Thus, inference-serving systems can exploit them to reduce SLO violations in the presence of interference without eviction or resource repartitioning. ODIN does not require offline resource utilization profiles for the inference, and relies only on runtime observed execution times of pipeline stages, therefore being easily applicable to any system. Additionally, ODIN avoids the costly process of building system-specific or pipeline-specific models to characterize interference. Instead, it dynamically reacts and adapts to the presence of interference while executing the inference pipeline. ODIN by itself does not have a notion of SLOs. It is a best-effort solution to quickly achieve near-optimal throughput and latency in the presence of interference, which thereby results in improved SLO conformance compared to a baseline least-loaded scheduler (LLS).

ODIN employs a heuristic pipeline scheduling algorithm, which uses the execution times of pipeline stages, compares them against interference-free performance values, and then moves network layers between pipeline stages, with the goal to reduce the work on the execution unit affected by interference, while maximizing the overall throughput of the pipeline. To minimize the duration of the mitigation phase and quickly react to performance changes due to interfer-

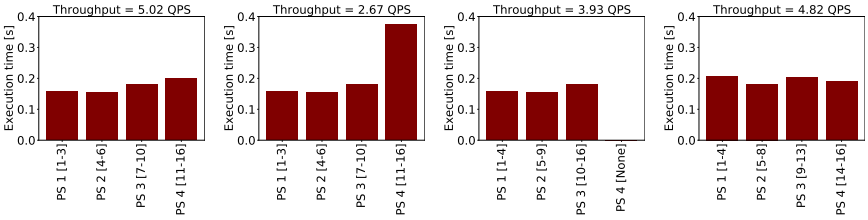
ence, the heuristic takes into account the extent of the performance degradation. We extensively test ODIN with 12 different scenarios of interference in 9 different frequency-duration settings and compare against the baseline least-loaded scheduler (LLS), which selects the least-loaded execution unit to assign work to. Our experiments show that ODIN sustains high throughput and low latency, including tail latency, under the different interference scenarios, and reacts quickly with a short mitigation phase, which takes 5-15 timesteps, outperforming LLS by 15% in latency and 20% in throughput on average. Additionally, with an SLO set at 80% of the original throughput, our solution is able to avoid 80% of SLO violations under interference, in contrast to LLS, which only delivers 50% SLO conformance. We also test the scalability of ODIN with a deep neural network model on highly parallel platforms, showing that the quality of the solution is independent of the number of execution units and depth of neural network.

2 Background and Motivation

Parallel inference pipelines provide a way to maximize the throughput of inference applications, as layer-wise parallelism offers reduced communication and minimizes the need to copy weights between execution units [2]. The parallelism exposed in parallel inference pipelines is across layers, with each layer being assigned to a pipeline stage, as well as within layers, where operators are parallelized for faster execution. A common way to execute pipelines is the "bind-to-stage" approach [18], where each stage of the pipeline is assigned to a unique set of compute units, i.e. an execution place, without sharing resources with other stages. In our work, we also assume that execution places do not share resources, therefore a pipeline stage will not experience interference from pipeline stages running on other execution places. To achieve high throughput, the pipeline stages need to be balanced, otherwise, throughput becomes limited by pipeline stalls, as the pipeline stages have a linear dependence.

Figure 1 shows a motivating example of an inference pipeline for VGG16, a CNN model. The pipeline consists of 4 stages, each consisting of 3 to 5 layers of the network model (Figure 1a), in a configuration where the pipeline stages are balanced in terms of execution time. Assuming a workload is colocated on the execution place which executes the fourth stage of the pipeline, the execution time of this stage increases due to interference, causing the throughput to decrease by 46% (Figure 1b). A static solution would dedicate the resources to the colocated workload, and would use only 3 execution places. To maintain high throughput, the pipeline stages would also be reduced to 3, leading to a suboptimal solution (Figure 1c). A dynamic solution would attempt to rebalance the initial four pipeline stages, to mitigate the effect of interference on the execution time of the fourth stage. An exhaustive search for an optimal new configuration is able to restore the initial throughput loss (Figure 1d), however this exhaustive search required 42.5 minutes to complete.

This experiment allows us to make the following observations: First, the effect of interference on a parallel inference pipeline can be mitigated by rebalancing the pipeline stages. Second, partitioning the resources between the colocated



(a) No interference (b) Interference on stage 4 (c) Execution with 3 stages (d) Exhaustive search rebalancing

Fig. 1: Throughput and execution time of a 4-stage pipeline for VGG-16.

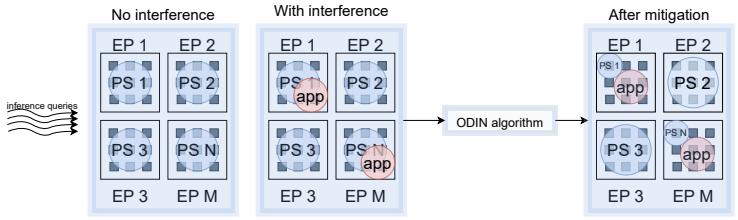


Fig. 2: System overview

workload and the inference pipeline leads to a shorted pipeline and a suboptimal throughput. Third, dynamic reaction to interference is able to largely restore throughput loss on the inference pipeline. Fourth, an exhaustive search for an optimal configuration is infeasible in a reactive, dynamic solution. The above observations motivate our work, which proposes an online scheduling technique for the pipeline stages of inference pipelines.

3 ODIN: A dynamic solution to overcome interference on inference pipelines

3.1 Methodology

In this work, we consider a system with a set of resources named execution places (EPs). Each execution place may consist of multiple cores, but execution places do not share performance-critical resources between them, e.g. caches, memory controllers/links. Inference pipelines are linear and are implemented with a bind-to-stage approach, where a single pipeline stage (PS) is assigned to a single EP, i.e. a unique set of resources of the system, and pipeline stages do not share resources. Pipeline stages can exploit the multiple resources within an EP by other means of parallelism, e.g. operator parallelism. A pipeline configuration defines the mapping of pipeline stages to execution places and the assignment of layers of a neural network model to PSs. We additionally assume that, in an interference-free system where the inference pipeline utilizes all the available execution places, the stages are already effectively balanced across the execution places. If a workload is colocated with a pipeline stage on one of the EPs, causing interference and increase of the execution time of this stage, the heuristics which form the backbone of our solution attempts to reduce the total work on the

affected pipeline stage, moving network layers to non-affected pipeline stages. A high-level overview of our approach, ODIN, is presented in Figure 2. Our approach operates online and is agnostic to any other colocated application. At runtime, we monitor the execution time of pipeline stages, and scan for changes in the performance of the slowest pipeline stage. If its execution time has increased, we consider it as affected by an interfering application and trigger the online rebalancing of pipeline stages, to find a new configuration, using our heuristic algorithm. If its execution time has decreased, we consider that any effect of interference is no longer present, and once again trigger online rebalancing to find a new configuration that reclaims resources from the colocated, interfering workload.

3.2 ODIN: A heuristic-based approach for pipeline stage re-balancing under interference

We describe our approach, ODIN, to mitigate the effect of interference on parallel inference pipelines, and the heuristics it uses to find new configurations for the pipeline stages at runtime. The complete steps of our approach are presented in Algorithm 1. The algorithm takes as input the current configuration C, which tracks the number of network layers belonging to each pipeline stage, and a tuning parameter α. As the algorithm starts operating without interference, the current configuration is considered to be optimal, and the pipeline throughput is the one given by the current configuration. During execution, the execution time of PSs is monitored. Interference is detected when the execution time t of one of the pipeline stages increases. We identify the affected PS ($\text{PS}_{affected}$) as the slowest stage in the current configuration, and this determines the throughput of the pipeline. The goal of the algorithm is then to rebalance the pipeline stages by removing layers from the affected PS, to reduce its work. We note that, removing layers from the affected PS may reduce the length of the pipeline by 1. We apply two heuristics to find a new configuration:

1) Set the direction for moving work: To remove layers from the affected PS, we first determine the direction of moving the layers. As the layers of an inference pipeline execute one after the other (forward pass), we can only remove layers from the head or tail of the $\text{PS}_{affected}$. At the first attempt, the algorithm does not know which layers of the $\text{PS}_{affected}$ have experienced performance degradation due to interference, so we initially remove layers from both ends, as shown in Lines 6-10, and move them to the preceding and subsequent pipeline stages respectively. Next, we calculate the sum of the execution time of PSs on both sides of the $\text{PS}_{affected}$ and set the direction to move layers. We then find the PS with the lowest execution time $\text{PS}_{lightest}$ in that direction, starting from $\text{PS}_{affected}$, and move one layer to $\text{PS}_{lightest}$, as shown in Lines 18-20.

2) Avoiding Local optimum Our first heuristic may result in a local, rather than a global optimum. A possible solution for this is to randomly choose a completely new starting configuration, and rebalance again. However, this can lead to loss of information. Since our initial configuration is optimal for the execution of the pipeline in an interference-free case, in the case of a local optimum, we deliberately move more layers from the $\text{PS}_{affected}$ to the $\text{PS}_{lightest}$, to create a different configuration and continue the exploration.

Algorithm 1 ODIN Algorithm

Require: C, α ▷ C = pipeline configuration
$T \leftarrow$ THROUGHPUT(C) ▷ T = throughput of the pipeline
$C_{opt} \leftarrow C$ ▷ Optimal pipeline configuration
$\gamma \leftarrow 0$ ▷ counter variable
while $\gamma < \alpha$ **do**
  $\mathrm{PS}_{affected} \leftarrow$ GET_INDEX$(\max(t(C)))$
  if $\gamma = 0$ **then**
    $C[\mathrm{PS}_{affected} + 1]$ += 1
    $C[\mathrm{PS}_{affected} - 1]$ += 1
    $C[\mathrm{PS}_{affected}]$ −= 2
  end if
  $\mathrm{S}_{left} \leftarrow$ SUM$(t(C[0], C[\mathrm{PS}_{affected}]))$
  $\mathrm{S}_{right} \leftarrow$ SUM$(t(C[\mathrm{PS}_{affected} + 1], C[N]))$
  if $\mathrm{S}_{left} < S_{right}$ **then**
    $direction \leftarrow left$
  else
    $direction \leftarrow right$
  end if
  $\mathrm{PS}_{lightest} \leftarrow$ GET_INDEX$(t(C, \mathrm{PS}_{affected}, direction))$
  $C[\mathrm{PS}_{affected}]$ −= 1
  $C[\mathrm{PS}_{lightest}]$ += 1
  $\mathrm{T}_{new} \leftarrow$ THROUGHPUT(C)
  if $\mathrm{T}_{new} < \mathrm{T}$ **then**
    γ += 1
  else if $\mathrm{T}_{new} = \mathrm{T}$ **then**
    $C[\mathrm{PS}_{affected}]$ −= 1
    $C[\mathrm{PS}_{lightest}]$ += 1
    γ += 1
  else
    $\gamma \leftarrow 0$
    $\mathrm{T} \leftarrow \mathrm{T}_{new}$
    $C_{opt} \leftarrow C$
  end if
end while
return C_{opt}

The extent of exploration is controlled by variable α which is provided as an input to the algorithm. As the algorithm is applied online, while the inference pipeline is running, the value of α can be tuned to reduce the number of trials for faster exploration. Figure 3 shows a timeline of an inference pipeline for VGG16, executing on four EPs with pipeline stages, where ODIN runs to mitigate the effects of interference. Initially, there is no interference, the inference pipeline is balanced with an optimal configuration and achieves its peak throughput. At time steps 5, 10, and 15, a new workload is co-located on a different execution place, slowing down the system for the inference pipeline, reducing what we define as the resource-constrained throughput, i.e. the throughput the inference pipeline can attain in the presence of interference. At each of these time steps, ODIN automatically detects the throughput degradation and rebalances the

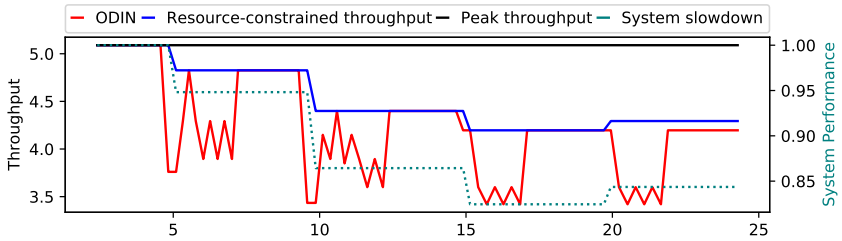


Fig. 3: A timeline of a VGG16 inference pipeline, running with ODIN, which reacts to mitigate interference at time steps 5, 10, 15, and 20.

pipeline until it finds a successful solution. At time step 20, one of the interfering workloads is removed, and ODIN executes again, to restore the pipeline throughput by claiming back the resources previously used by the colocated workload.

3.3 Implementation details

Database Creation: In our evaluation, we use simulation to be able to apply ODIN on any type and size of the underlying system. We, therefore, replace online monitoring with an offline database. We first collect the execution time of the m individual network layers of the inference pipelines under consideration, when executing alone (without any interference), on a real platform. On the same platform, we collect the execution time of the individual network layers when executing alongside co-located applications, producing n different interference scenarios. We then store these collected $m \times (n+1)$ measurements in a database, and use them in simulation. We consider the real platform to be a single execution place for ODIN, and simulate multiple execution places of the same type. To emulate interference, during simulation, we randomly select an interference scenario for an execution place and look up the corresponding execution time in the database.

Throughput calculation: We use the measurements in our database D of size $m \times n$ to calculate the throughput of a pipeline, as follows:

$$T = \frac{1}{max_{i=0}^{N} \sum_{l=0}^{P} D[l,k]}$$

where N is the number of pipeline stages, P is the number of layers in a pipeline stage, and $D[l,k]$ is the execution time of layer l under the type of interference k, as recorded in the database D.

Implementation of the least-loaded scheduler (LLS) as a baseline: LLS is an online interference mitigation technique [9,11,26]. We implement LLS in the context of pipeline stages, as a baseline to compare against ODIN. We calculate the utilization of each pipeline stage and move the layers from the most utilized to the least utilized stage recursively until the throughput starts decreasing. The utilization of a stage v_i is calculated as:

$$v_i = \left(1 - \frac{w_i}{w_i + t_i}\right)$$

where t_i is the execution time of a pipeline stage, and w_i is the waiting time of the stage, calculated as $w_i = w_{i-1} + ti - 1 - t_i$, with $w_0 = 0$.

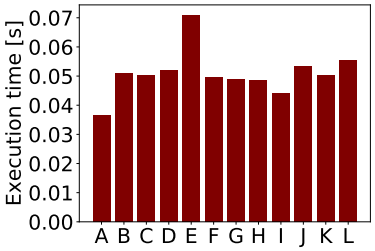


Fig. 4: Performance impact

Mode of execution	Core assignment on Alder Lake
A	CNN:[0-7]
B	CNN: [0-7], IBench-MemBW: [0]
C	CNN: [0-7], IBench-MemBW: [0-1]
D	CNN: [0-7], IBench-MemBW: [0-3]
E	CNN: [0-7], IBench-MemBW: [0-7]
F	CNN: [0-7], IBench-CPU: [0]
G	CNN: [0-7], IBench-CPU: [0-1]
H	CNN: [0-7], IBench-CPU: [0-3]
I	CNN: [0-7], IBench-CPU: [0-7]
J	CNN: [0-3], IBench-MemBW: [4-7]
K	CNN: [0-3], IBench-CPU: [4-7]
L	CNN: [0-3], IBench-CPU: [4-7], IBench-MemBW[4-7]

Table 1: Interference scenarios

4 Evaluation

4.1 Experimental setup

We execute ODIN in a simulated system for inference serving, which consists of multiple execution places, and each execution place consists of a fixed number of 8 cores. To generate our database, we use an Intel i9-12900K (AlderLake) server, which consists of 8 2xP-cores (Performance) and 8 2xE-cores (Efficient). We consider the set of 8 P-cores as a single execution place in our system.

For the neural network models we examine as inference pipelines, our database consists of measurements for each layer without interference, as well as measurements for each layer with 12 different co-located workloads, in different settings. To create the co-located workloads, we use two interference benchmarks from the `iBench` suite [8], the `CPU` benchmark that stresses the CPU and the `memBW` benchmark that stresses the memory bandwidth. We then create our 12 scenarios of colocation by assigning the network layers and interference benchmarks different numbers of threads, and pinning them to different cores. Table 1 showcases the colocation scenarios considered in our database, and Figure 4 demonstrates the performance impact of interference for all these colocation scenarios on a single layer of the VGG16 network model.

For our evaluation, we consider the inference pipelines of three popular CNN models: VGG16 [27], ResNet-50 and ResNet-152 [13], with 16, 50, and 152 layers respectively, implemented with the Keras [6] framework.

4.2 Interference mitigation with ODIN

To evaluate the effectiveness of ODIN, we compare its latency and throughput for different values of α, which sets the extent of exploration, against LLS, in several interference scenarios. In particular, we consider a system of 4 executions places of 8 cores each, which serves inference queries with two network models, VGG16, and ResNet-50. We assume a fixed number of 4000 queries, and induce random interference on different execution places, based on the colocation scenarios described in Table 1. We consider different values for the frequency (frequency periods of 2, 10, and 100 queries) and duration (2, 10, and 100 queries) of interference, and evaluate the end-to-end latency and throughput distribution of each inference pipeline.

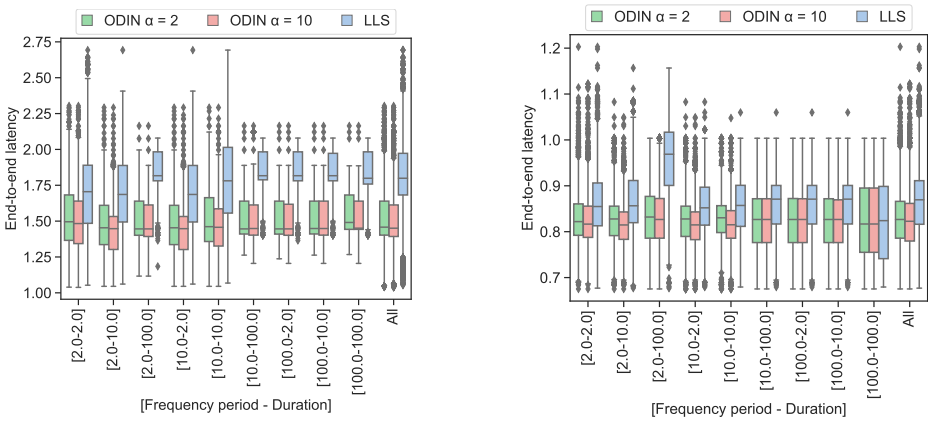
Latency: Figure 5 shows the latency distribution of the two inference pipelines under interference. We observe that ODIN outperforms LLS in all scenarios, delivering lower latency. We highlight the effect of the α parameter of ODIN on

latency. A higher value of α yields lower latency, because the longer exploration phase allows ODIN to find an optimal configuration. On the other hand, if the frequency of interference is high, a low value of α is able, in most cases, to produce an equally good solution with lower exploration time. ODIN $\alpha = 10$ yields better latency than ODIN $\alpha = 2$ this is because the former takes more trials to find a schedule, however if the frequency of interference is high then it may take longer to find a solution or end up with sub-optimal solution. We additionally note that both ODIN and LLS are more effective in cases where interference appears with lower frequency and for longer periods. This is particularly evident in Figure 5. For the pair of [frequency period = 2, duration = 2], the distribution of latency shows many outliers, as an optimal configuration found by the algorithm for one period of interference may be applied to the next period, where the pattern of interference has changed. Overall, however, ODIN outperforms LLS in all scenarios, offering 15.8% better latency on average with $\alpha = 10$ and 14.1% with $\alpha = 2$.

Throughput: We then compare the throughput of the inference pipelines under interference, for ResNet50 and VGG16, with ODIN and LLS, for the same interference scenarios, in Figure 6. Again, ODIN offers higher throughput than LLS in most cases. The case of VGG16 highlights our observation about the lower performance in the case of high frequency, where all three techniques show outliers of low throughput, however, ODIN is more able to adapt to interference of longer duration compared to LLS. We observe additionally that for the case of the highest frequency period-duration pair [100, 100], LLS and ODIN have comparable performance, as the near-optimal solutions were obtained with minimal changes of the pipeline configurations. Overall, on average, ODIN achieves 19% higher throughput than LLS with any choice of α.

Tail latency: Besides the latency distribution, we separately examine the tail latency (99th percentile), as it can be a critical metric in inference-serving systems, and it is also indicative of the quality of the solutions found by ODIN. Figure 7 shows the distribution of the tail latency across all the queries considered in the interference scenarios examined in this Section. For both ResNet50 and VGG16, ODIN results in significantly lower tail latencies than LLS. For the case of VGG16, we additionally observe that a higher value of α for ODIN can produce better solutions, resulting in lower tail latencies. On average, ODIN results in 14% lower tail latencies than LLS.

Exploration overhead: Upon detection of interference, both ODIN and LLS begin the rebalancing phase, during which queries as processed serially, until a new configuration of the pipeline stages is found. On average, the number of queries that will be processed serially during a rebalancing phase is 1 for LLS, and 4 and 12 for ODIN with $\alpha = 2$ and $\alpha = 10$ respectively. Figure 8 shows the percentage of time required to rebalance the pipeline stages, for the window of 4000 queries. It is evident that, if the type of interference changes frequently and is short-lived, the overhead of ODIN is higher, as the system is almost continuously in a rebalancing phase. However, when the duration of interference is longer, as the effect of interference on the inference pipeline may be the same, rebalancing may not be triggered, as the selected configuration is already optimal,



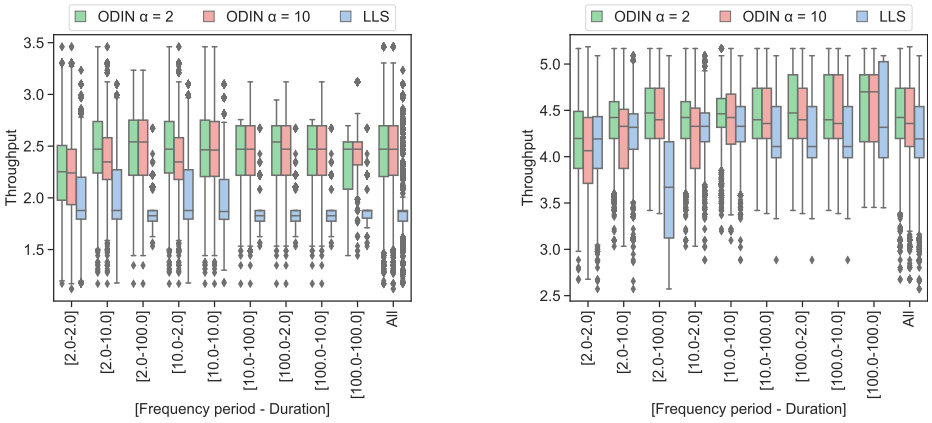
(a) ResNet50 (b) VGG16

Fig. 5: Inference pipeline latency (lower is better) with ODIN, in comparison to LLS, over a window of 4000 queries, for interference of different frequency period and duration.

therefore the rebalancing overhead decreases. Longer frequencies and durations of interference are favored by both ODIN and LLS.

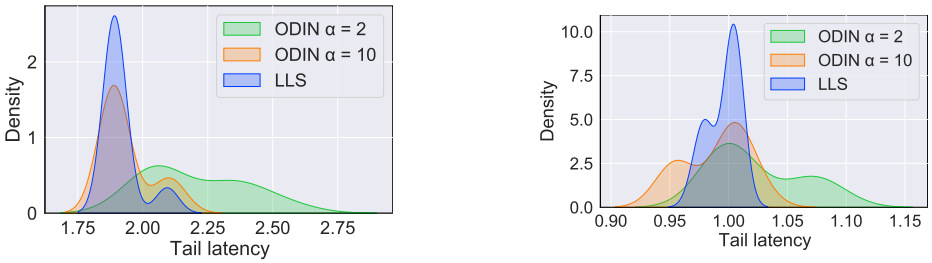
4.3 Maintaining QoS with ODIN

To evaluate the ability of ODIN to mitigate interference on an inference pipeline, we consider its quality-of-service (QoS) in terms of SLO violations [1,25]. We use throughput as the target QoS metric, and consider the SLO level as the percentage of the peak throughput, i.e. the throughput of the inference pipeline when executing alone. We then profile the number of queries which violate this SLO using ODIN and LLS. We additionally compare the SLO violations with respect to the resource-constrained throughput, i.e. the throughput achieved when a colocated workload causes interference, and an optimal configuration of the pipeline is found through exhaustive search. We present the results in Figure 9. Although neither ODIN or LLS are able to offer any performance guarantees, resulting in many violations when the SLO level is strict, ODIN results in less than 20% of SLO violations for SLO levels lower than 85%, and can sustain 70% of the original throughput for any interference scenario, in contrast to LLS, which, in the extreme case of VGG16, violates even an SLO of 35% of the original throughput. Additionally, the comparison of SLO violations for the SLO set w.r.t. the resource-constrained throughput shows that ODIN is able to find near-optimal configurations in most cases, which are close to those found by the exhaustive search. Our conclusion is that, while ODIN cannot provide any strict guarantee for a set SLO, it can sustain high throughput under looser SLOs and therefore can be an effective solution for overprovisioned systems. For example, an inference-serving system that can tolerate 10% of SLO violations would require to overprovision resources by 42% with ODIN, compared to 150% for LLS.



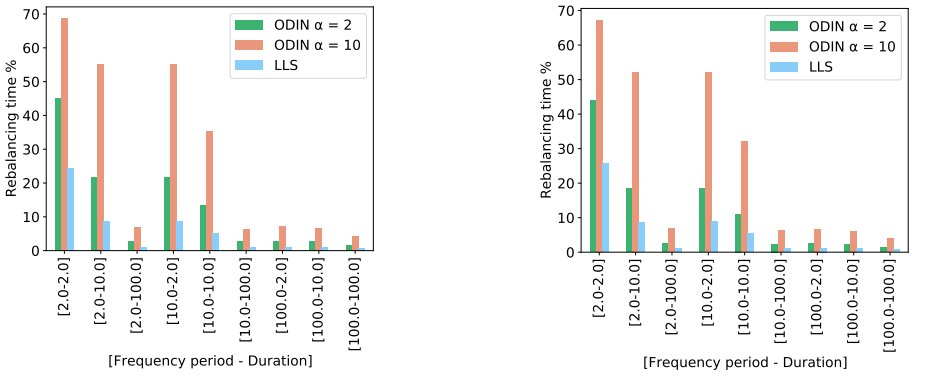
(a) ResNet50 (b) VGG16

Fig. 6: Inference pipeline throughput (higher is better) with ODIN, in comparison to LLS, over a window of 4000 queries, for interference of different frequency period and duration.



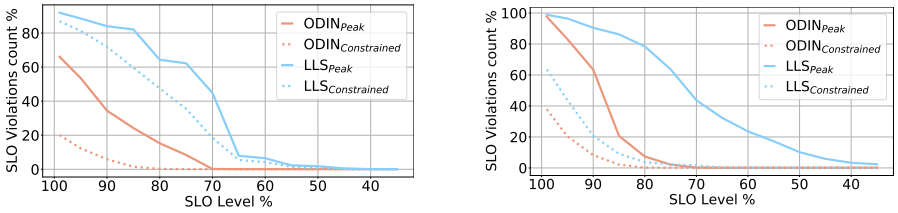
(a) ResNet50 (b) VGG16

Fig. 7: Tail latency distribution of ODIN, in comparison to LLS.



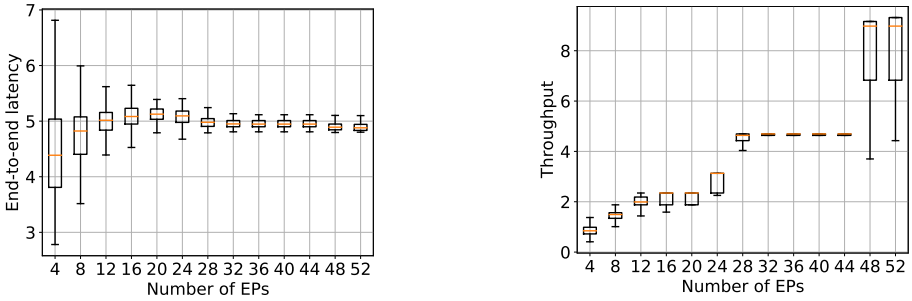
(a) ResNet50 (b) VGG16

Fig. 8: Overhead analysis of ODIN, in comparison to LLS.



(a) ResNet50 (b) VGG16

Fig. 9: Quality-of-service of ODIN, in comparison to LLS, for different SLO levels.



(a) Latency (b) Throughput

Fig. 10: Scalability analysis of ODIN with ResNet152.

4.4 Scalability analysis of ODIN

We finally analyze the scalability of ODIN on high numbers of execution places, with deep network models that can run with multiple pipeline stages. For this, we use ResNet152, which consists of 152 layers. We consider, however, residual blocks as a single unit, so the maximum number of pipeline stages ResNet152 could run with is 52. We scale the number of execution places from 4 up to 52, and consider a window of 4000 queries, with interference of a frequency period of 10 and duration of 10 queries. Figure 10 shows the latency and throughput of ODIN for the different numbers of EPs. The latency is not affected as the number of EPs increases, therefore ODIN is effective at finding optimal pipeline configurations on multiple execution places. Equivalently, throughput increases with the number of EPs, suggesting high parallelism of the pipeline, and for 52 EPs, the achieved throughput is comparable to the peak throughput of the inference pipeline, under no interference.

5 Conclusion

In this work, we have proposed ODIN, an online pipeline rebalancing technique that mitigates the effect of interference on inference pipelines. ODIN utilizes the execution times of the pipeline stages to readjust the assignment of layers to pipeline stages, according to the available resources, rebalancing the pipeline. We show that ODIN outperforms the baseline LLS in latency and throughput under different interference scenarios. Additionally, ODIN maintains more than 70% of the peak throughput of the pipeline under interference, and

achieves very low SLO violations compared to LLS. Finally, ODIN scales well with deeper networks and large platforms. ODIN is online and dynamic, and requires minimal information from the inference pipeline, therefore applies to any type of inference pipeline and interference scenario. The abstraction of the hardware into execution places allows ODIN to be applied to different types of hardware platforms. As future work, we plan to parallelize the pipeline during rebalancing, and validate the utility of ODIN on heterogeneous platforms.

Acknowledgement: This work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048. The computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at NSC, partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

References

1. Alves, M.M., Teylo, L., Frota, Y., Drummond, L.M.d.A.: An interference-aware strategy for co-locating high performance computing applications in clouds. In: Symposium on High Performance Computing Systems. pp. 3–20. Springer (2020)
2. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) (2019)
3. Chang, H.Y., Mozafari, S.H., Chen, C., Clark, J.J., Meyer, B.H., Gross, W.J.: Pipebert: High-throughput bert inference for arm big. little multi-core processors. Journal of Signal Processing Systems pp. 1–18 (2022)
4. Chen, Q., Yang, H., Guo, M., Kannan, R.S., Mars, J., Tang, L.: Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In: Proceedings of the 22nd ASPLOS'17. pp. 17–32
5. Chen, Q., Yang, H., Mars, J., Tang, L.: Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. ACM SIGPLAN Notices **51**(4), 681–696 (2016)
6. Chollet, F., et al.: Keras. `https://keras.io` (2015)
7. Crankshaw, D., Wang, X., Zhou, G., Franklin, M.J., Gonzalez, J.E., Stoica, I.: Clipper: A low-latency online prediction serving system. In: NSDI (2017)
8. Delimitrou, C., Kozyrakis, C.: ibench: Quantifying interference for datacenter applications. In: IISWC. pp. 23–33. IEEE (2013)
9. Delimitrou, C., Kozyrakis, C.: Paragon: Qos-aware scheduling for heterogeneous datacenters. ACM SIGPLAN Notices **48**(4), 77–88 (2013)
10. Delimitrou, C., Kozyrakis, C.: Quasar: Resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices **49**(4), 127–144 (2014)
11. Devi, D.C., Uthariaraj, V.R.: Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks. The scientific world journal **2016** (2016)
12. Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al.: Dapple: A pipelined data parallel approach for training large models. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 431–445 (2021)
13. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
14. Huang, et al.: Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems **32** (2019)

15. Jeong, E., Kim, J., Tan, S., Lee, J., Ha, S.: Deep learning inference parallelization on heterogeneous processors with tensorrt. IEEE Embedded Systems Letters **14**(1), 15–18 (2021)
16. Kang, D., Oh, J., Choi, J., Yi, Y., Ha, S.: Scheduling of deep learning applications onto heterogeneous processors in an embedded device. IEEE Access **8** (2020)
17. Ke, L., Gupta, U., Hempsteadis, M., Wu, C.J., Lee, H.H.S., Zhang, X.: Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In: HPCA. pp. 141–144. IEEE (2022)
18. Lee, I.T.A., Leiserson, C.E., Schardl, T.B., Zhang, Z., Sukha, J.: On-the-fly pipeline parallelism. ACM Transactions on Parallel Computing (TOPC) **2**(3), 1–42 (2015)
19. Lee, Y., Scolari, A., Chun, B.G., Santambrogio, M.D., Weimer, M., Interlandi, M.: Pretzel: Opening the black box of machine learning prediction serving systems. In: OSDI. vol. 18, pp. 611–626 (2018)
20. Li, S., Hoefler, T.: Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14 (2021)
21. Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., et al.: Elastic machine learning algorithms in amazon sagemaker. In: SIGMOD. pp. 731–737 (2020)
22. Mendoza, D., Romero, F., Li, Q., Yadwadkar, N.J., Kozyrakis, C.: Interference-aware scheduling for inference serving. In: Proceedings of the 1st Workshop on Machine Learning and Systems. pp. 80–88 (2021)
23. Narayanan, et al.: Pipedream: Generalized pipeline parallelism for dnn training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 1–15 (2019)
24. Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., Soyke, J.: Tensorflow-serving: Flexible, high-performance ml serving. arXiv preprint arXiv:1712.06139 (2017)
25. Romero, F., Li, Q., Yadwadkar, N.J., Kozyrakis, C.: Infaas: Automated model-less inference serving. In: USENIX Annual Technical Conference. pp. 397–411 (2021)
26. Shaw, S.B., Singh, A.: A survey on scheduling and load balancing techniques in cloud computing environment. In: ICCCT. pp. 87–95. IEEE (2014)
27. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
28. Soomro, P.N., Abduljabbar, M., Castrillon, J., Pericàs, M.: An online guided tuning approach to run cnn pipelines on edge devices. In: CF'21
29. Soomro, P.N., Abduljabbar, M., Castrillon, J., Pericàs, M.: Shisha: Online scheduling of cnn pipelines on heterogeneous architectures. In: PPAM 2022
30. Wang, et al.: High-throughput cnn inference on embedded arm big. little multicore processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(10), 2254–2267 (2019)
31. Yeung, G., Borowiec, D., Yang, R., Friday, A., Harper, R., Garraghan, P.: Horus: An interference-aware resource manager for deep learning systems. In: ICA3PP. pp. 492–508. Springer (2020)
32. Zhang, C., Yu, M., Wang, W., Yan, F.: Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In: USENIX Annual Technical Conference. pp. 1049–1062 (2019)