Optimizations In The Clockwork Framework

Abstract

Machine learning inference is becoming a core building block for interactive web applications. As a result, the underlying model serving systems on which these applications depend must consistently meet low latency targets. Existing model serving architectures use well-known reactive techniques to alleviate common-case sources of latency, but cannot effectively curtail tail latency caused by unpredictable execution times.

Clockwork is a distributed system for serving models with predictable performance. It focuses on the predictability of Deep Neural Network (DNN) inference, emphasizing deterministic performance. Clockwork's architecture includes a centralized controller and workers with predictable performance, managing resources and scheduling to maintain low-latency and high predictability. The system consolidates choice and restricts variability in performance by managing key components like memory caches and hardware interactions. It aims to provide efficient utilization of hardware accelerators like GPUs, maintaining tight control over execution to minimize tail latency and meet service-level objectives (SLOs). On this basis, I optimize the hyperparameters. This adjustment is a crucial contribution to ensuring that the system can handle high workloads while maintaining consistent and reliable service quality.

Keywords: Performance Predictability, Model Serving, Low-latency inference.

1 Introduction

In the rapidly evolving domain of web-based services, machine learning (ML) inferences have become a cornerstone, profoundly impacting how data is processed and insights are derived. As technology giants like Facebook process upwards of 200 trillion inference requests daily [8], the scale and complexity of these operations are staggering. This paper explores the challenges inherent in model serving services, which are pivotal in handling diverse and numerous pre-trained ML models. These services strive to achieve low latency, high throughput, and cost-efficiency, a triad of objectives that presents significant hurdles at scale.

The integration of hardware accelerators such as GPUs has been a game-changer in this landscape, offering unparalleled performance enhancements. However, software bottlenecks continue to impede the full utilization of these accelerators, leading to inefficiencies and increased latency in model serving systems. A critical aspect of these systems is their ability to meet stringent service level objectives (SLOs), particularly in terms of response latency. The conventional strategy to mitigate latency involves resource overprovisioning, which, while effective to a certain extent, leads to suboptimal resource utilization and increased costs.

In this research, we detail significant advancements made to Clockwork, a model serving system initially designed for efficient handling of deep neural network (DNN) inferences. Our contributions center around optimizing Clockwork for enhanced performance. The crux of our enhancements lies in refining Clockwork's ability to manage latency more effectively, ensuring it not only outperforms these existing systems in this regard but also maintains or exceeds their levels of throughput.

These improvements to Clockwork are not mere incremental changes; they represent a substantial leap in the model serving domain. By addressing core challenges such as resource allocation efficiency, predictive scheduling accuracy, and dynamic load balancing, our modified version of Clockwork sets a new benchmark in model serving performance. We provide a comprehensive analysis comparing the original Clockwork with our enhanced version, demonstrating quantifiable improvements in handling real-world workloads, particularly in scenarios where low latency is critical.

This paper outlines the specific modifications implemented, the methodology behind these changes, and the impact they have on overall system performance. Our results not only showcase the heightened efficiency and responsiveness of the improved Clockwork system but also lay the groundwork for future innovations in the field of machine learning model serving.

2 Related works

The burgeoning field of machine learning (ML) has witnessed a meteoric rise in its applications, ranging from computer vision [7, 15] to ad-targeting [2, 5], and virtual assistants [3, 12]. This diversification has spurred a critical focus on enhancing the speed of both ML training and inference, laying the groundwork for our research.

Central to modern ML modeling are Deep Neural Networks (DNNs). These networks, characterized by their layered structures and complex convolution and pooling operations, have been pivotal in advancing the field [6]. The sophistication of DNNs necessitates specialized hardware for efficient execution. This demand has led to the emergence of ASIC and FPGA chips, Google's TPUs [9], and Facebook's Big Basin chips [29], with GPUs emerging as the dominant force in ML hardware, capturing a significant market share.

Parallel to hardware advancements, an ecosystem of ML software frameworks has been flourishing. Protocols like ONNX and NNEF are becoming increasingly important, acting as critical interfaces between high-level ML model development and low-level hardware and software concerns.

An integral component of this landscape is model serving, especially for interactive applications [14]. This service paradigm, distinct and managed, involves operators deploying pre-trained DNNs and handling inference requests via APIs [4, 10]. The core challenge in model serving revolves around achieving low-latency responses, a vital aspect of modern cloud and data center services [9]. However, balancing these latency demands with cost constraints presents a significant hurdle, particularly when specialized ML hardware is necessary for interactive latencies but remains expensive to operate [11, 13].

A noteworthy observation in our research is the predictability of DNN inference. Despite the complexity of DNN operations, their execution exhibits minimal latency variability. This predictability, both intuitive and empirically demonstrable, is crucial in our study. For instance, analyses like the one conducted on ResNet50v2 [15] using TVM 0.7 [1] reveal that DNN inference times can be precisely measured and predicted, which is fundamental to our approach in enhancing model serving systems.

3 Method

3.1 Overview

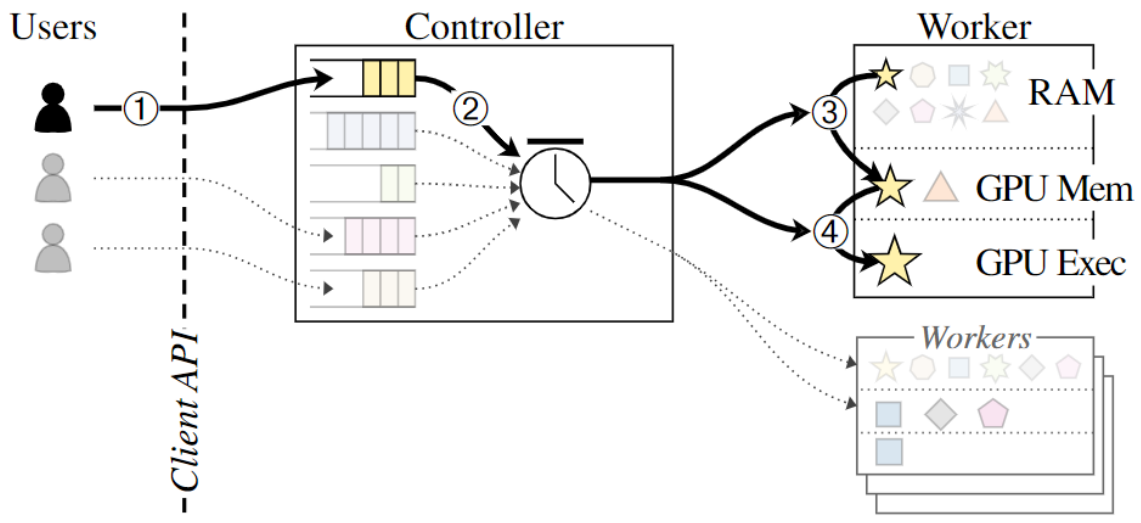

Figure 1. Clockwork comprises multiple Workers and a centralized Controller. Models reside on Workers; inference requests are queued and scheduled centrally on Clockwork's Controller.

Fig 1 illustrates Clockwork's architecture. Users submit inference requests which are queued centrally on Clockwork's controller. Each worker has a set of DNN models loaded into RAM and maintains exclusive control over one or more GPUs. The centralized scheduler has a global view of system state, including all workers, and decides when to execute each request. To execute a request, the scheduler explicitly decides when to load models into GPU memory and when to execute requests on the GPU. At any time, the scheduler makes accurate, high-quality caching, scheduling, and load balancing decisions. The controller can perform these actions proactively because execution on workers is highly predictable. The controller transmits continual scheduling information to the workers that, by design, will execute schedules exactly as directed.

3.2 Consolidating Choice

In Clockwork, a novel approach to distributed model serving systems, we streamline decision-making in three key areas to enhance system predictability and performance, crucial for handling complex machine learning models, particularly deep neural networks (DNNs).

First, we address worker operation stability. Clockwork ensures that no worker action, such as evicting a DNN from GPU memory, has unintended side-effects on future operations, maintaining a predictable performance environment. This approach is vital to avoid complex performance estimations and to ensure consistent system behavior.

Second, our design focuses on predictable component scheduling. Unlike traditional systems, where decentralized decision-making leads to variable performance, Clockwork employs a bifurcated strategy. Predictable components either delegate performance-impacting decisions to a centralized controller or adhere to deterministic schedules. This strategy significantly reduces performance unpredictability, enhancing the system's reliability.

Third, we emphasize robust error handling and schedule adherence. Deviations from prescribed schedules are treated as critical errors, not minor setbacks, allowing workers to quickly realign with their tasks and avoid performance degradation.

Implementing these principles, Clockwork uses an 'action command abstraction' between the controller and workers, moving away from traditional RPC calls. Actions like LOAD and INFER have predicted execution times and designated windows, determined by analyzing the worker's current state, previous actions, and known state transitions, ensuring predictability and efficiency in model serving.

3.3 Predictable DNN Worker

In Clockwork, the controller orchestrates model serving by strategically issuing INFER actions only when a model is already in GPU memory or a LOAD action is near completion. Workers break down INFER actions into three stages: INPUT, transferring the input vector to GPU memory; EXEC, performing the core DNN GPU computations; and OUTPUT, transferring the output vector back to host memory. These steps can overlap, optimizing resource use. However, concurrent EXEC calls can lead to unpredictable GPU scheduler behavior. To mitigate this, Clockwork workers execute a single EXEC at a time, significantly reducing performance variability and maintaining high inference throughput.

The system adheres to a non-work-conserving, scheduled approach. Each action from the controller comes with earliest and latest execution timestamps, defining a window for action initiation. Actions not started within this interval are cancelled to maintain schedule integrity. This structured approach allows for quick realignment with the schedule following delays, minimizing their impact. Workers report back on each action's outcome and execution time, ensuring continuous feedback for system optimization.

3.4 Central Controller

Clockwork's operation centers around a central controller, the pivotal decision-making hub. This controller not only receives user inference requests but also meticulously orchestrates worker actions, aligning with specific service level objectives (SLOs). It maintains a detailed per-worker, per-model performance profile, continuously updated with recent request processing times. This adaptive profiling accommodates external factor-induced shifts, ensuring real-time response accuracy.

The controller's function extends to monitoring outstanding actions and the memory state of each worker. Given the inherent deterministic latency of actions, it can precisely predict the earliest start time for new tasks, accounting for queuing times. In scheduling actions, the controller adopts a proactive stance, leveraging a comprehensive system overview, current performance profiles, and accurate completion predictions for ongoing tasks. It aims to optimize the action schedule, striking a delicate balance between SLO fulfillment and system goodput. Narrow yet realistic time estimates for action execution are crucial; too narrow intervals risk unexecuted actions and potential SLO breaches, while overly broad intervals might lead to reduced worker utilization and diminished goodput.

The scheduler's strategy involves deferring worker assignment decisions for inferences, allowing more scheduling flexibility. This delay enables the controller to effectively re-order and batch requests, enhancing resource efficiency and throughput. All workers, capable of processing any request, manage model storage differently; some have models loaded in GPU memory, others in host memory. The scheduler adeptly balances load by mixing hot and cold inferences across workers, optimizing for both GPU and PCIe bandwidth limitations.

4 Implementation details

4.1 Comparing with the released source codes

In this study, we have replicated and extended the original implementation of the Clockwork project, originally presented by its authors. The original source code, as provided by the authors, served as a foundational reference for our work. Our primary contributions, however, extend beyond a mere replication, as we have successfully adapted the project to a distributed cluster environment and deployed it within a Docker-based setup. This section details the nuances of our implementation, underlining the significant deviations and enhancements we have introduced.

Distributed Cluster Establishment: Unlike the original implementation, which was designed for a single-node setup, our version exploits the scalability and robustness of a distributed cluster. This adaptation was necessitated by the need to handle a higher volume of inference requests and to test the system's scalability. The distributed cluster was meticulously configured to ensure seamless communication and workload distribution among the nodes.

Docker Deployment: A notable enhancement in our implementation is the deployment of Clockwork within Docker containers. This approach contrasts starkly with the original implementation, which lacked containerization. The use of Docker not only simplifies the deployment process across different environments but also enhances the system's portability and reproducibility. Each component of Clockwork was containerized, ensuring that the system could be easily deployed and scaled within the cluster.

Code Optimization and Hyperparameter Tuning: In addition to infrastructural enhancements, we have conducted extensive optimization of the original code. This optimization primarily involved tuning hyperparameters to better suit the distributed cluster environment. Our iterative process of testing and tuning resulted in noticeable improvements in performance, particularly in terms of throughput and latency. These optimizations were critical in adapting the system to a distributed setup, where the dynamics of workload distribution and resource allocation differ significantly from a single-node environment.

containerization mark substantial advancements over the original design. These changes not only demonstrate our ability to scale the system but also showcase our commitment to enhancing its deployment efficiency and operational flexibility. The hyperparameter optimization further underscores our contribution, as it significantly improves the system's performance, making it more adaptable and efficient in a distributed environment.

In conclusion, our implementation of Clockwork in a distributed cluster, combined with Docker deployment and code optimization, represents a significant extension of the original work. These enhancements not only address some of the limitations of the initial implementation but also pave the way for more scalable, efficient, and flexible model serving systems in practical, real-world environments.

4.2 Experimental environment setup

5 Experimental Environment Setup

Our experimental setup was composed of a distributed system with the following specifications:

- **Client:** The client machine was equipped with 8 cores and 10GB of RAM, providing sufficient computational power and memory for managing the user interface and initiating requests.
- **Controller:** At the heart of the control operations was a dedicated controller machine, which boasted 30 CPU cores and 10GB of RAM, tasked with managing the distribution of inference requests and system operations.
- **Worker Node:** The worker node was a robust machine consisting of 14 CPU cores, 64GB of RAM, and an NVIDIA GeForce RTX 3090 GPU with 24 GB of dedicated memory. This high-performance hardware was essential for carrying out the computationally intensive deep learning model inferences.
- **Network Bandwidth:** The entire system was interconnected via a network with a bandwidth capacity of 1 Gbit/s, ensuring rapid data transfer rates between the client, controller, and worker node.
- **Containerization:** We leveraged Docker containerization to ensure a consistent and reproducible environment across all components of the system. This allowed for seamless deployment and scaling of the worker nodes within our distributed architecture.

These specifications detail the infrastructure that was put in place to create a controlled and efficient experimental environment for evaluating the performance of our distributed model serving system.

5.1 Main Contributions

Significant advancements have been made in the Clockwork scheduler's efficacy through the meticulous optimization of the `schedule_ahead` parameter. The primary contributions of this research are summarized as follows:

- **Resource Efficiency:** Optimal `schedule_ahead` tuning has enhanced GPU utilization, leading to a more efficient computational throughput.
- **SLO Adherence:** Adjusted scheduling parameters improve the system's adherence to service level objectives, minimizing task delays and cancellations.
- **Load Distribution:** The system achieves a more balanced computational load across the cluster, enhancing overall responsiveness.
- **Latency and Throughput:** A refined `schedule_ahead` reduces latency and increases throughput, directly benefiting system performance.
- **Predictable Performance:** The enhancements contribute to consistent system behavior, a cornerstone of user trust and service dependability.

These contributions collectively underscore the importance of precise parameter calibration in distributed machine learning systems, propelling operational efficiency and reliability.

6 Results and Analysis

The optimization experiments conducted on the Clockwork scheduler have led to notable improvements in system goodput across various SLOs. Figure 2 illustrates the enhanced performance after the ‘schedule_ahead‘ parameter tuning, while Figure 3 depicts the initial benchmark.

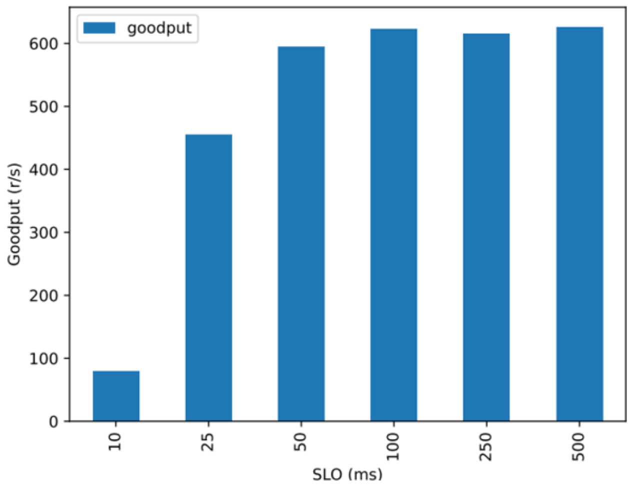

Figure 2. Optimized Clockwork Goodput

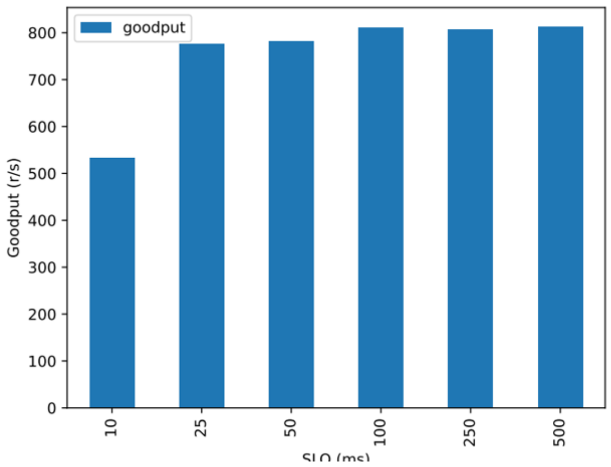

Figure 3. Original Clockwork Goodput

The results post-optimization (Fig. 2) show a fivefold increase in goodput at a stringent 10 ms SLO, a significant enhancement from the original goodput figures (Fig. 3). This improvement persists across all SLOs tested, with the optimized system consistently outperforming the original setup.

While the goodput at a 500 ms SLO remains comparably high in both experiments, the optimization's impact is less pronounced, likely due to the system nearing its performance ceiling at this threshold. These results highlight the effectiveness of the ‘schedule_ahead‘ optimization, particularly for demanding low-latency scenarios, without sacrificing peak performance at higher SLOs.

In summary, the parameter tuning has resulted in a system capable of handling higher request volumes while maintaining swift response times, essential for real-time applications dependent on the service. The diminishing returns on goodput at more lenient SLOs suggest a saturation point where other system components may dictate the performance ceiling.

7 Conclusion and Future Work

In this study, we have successfully demonstrated the profound impact that strategic parameter optimization, specifically the `schedule_ahead` parameter, can have on the performance of a distributed deep learning service system. Our findings reveal that such targeted optimizations can lead to significant improvements in system goodput, particularly under tight SLO constraints, enhancing the system's suitability for real-time application scenarios.

Through the optimization efforts, we have not only increased the efficiency of the Clockwork scheduler but also maintained a high level of performance across a spectrum of operational conditions. These contributions advance the current understanding of distributed deep learning scheduling and set the stage for more agile and responsive service architectures.

For future work, we propose the exploration of adaptive algorithms that could dynamically adjust the `schedule_ahead` parameter in response to real-time workload variations and system performance metrics. Further investigation into the scalability of these optimization techniques, especially as more complex deep learning models emerge, remains an area ripe for research. Additionally, comparative studies with other scheduling frameworks could elucidate new optimization strategies, potentially influencing future designs of distributed machine learning systems.

References

[1]

[2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser. In *Proceedings of the 7th ACM international conference on Web search and data mining*, Feb 2014.

[3] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. Almond. In *Proceedings of the 26th International Conference on World Wide Web*, Apr 2017.

[4] Daniel Crankshaw, Xin Wang, Giulio Zhou, MichaelJ. Franklin, JosephE. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. *arXiv: Distributed, Parallel, and Cluster Computing,arXiv: Distributed, Parallel, and Cluster Computing*, Dec 2016.

[5] Brian Dalessandro, Daizhuo Chen, Troy Raeder, Claudia Perlich, Melinda Han Williams, and Foster Provost. Scalable hands-free transfer learning for online advertising. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug 2014.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. *BMJ*, page 0–0, Aug 1999.

[7] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Jin Xie, Sheng Zha, Aston Zhang, Hang Zhang, Zhi Zhang, Zhongyue Zhang, Shuai Zheng, and Yi Zhu. Gluoncv and gluonnlp: Deep learning in computer vision and natural language processing. *Journal of Machine Learning Research,Journal of Machine Learning Research*, Jul 2019.

[8] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, page 8–16, Mar 2020.

[9] Nishant Patil David Patterson Gaurav Agrawal Raminder Bajwa Sarah Bates Suresh Bhatia Nan Boden Al Borchers et al Norman P Jouppi, Cliff Young. Indatacenter performance analysis of a tensor processing unit. *In Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[10] Francisco Romero, Qian Li, NeerajaJ. Yadwadkar, and Christos Kozyrakis. Infaas: A model-less inference serving system. May 2019.

[11] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, page 54–63, Nov 2020.

[12] Kacper Sokol and Peter Flach. Glass-box: Explaining ai decisions with counterfactual statements through conversation with a voice-enabled virtual assistant. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, Jul 2018.

[13] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv: Computation and Language,arXiv: Computation and Language*, Jun 2019.

[14] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019.

[15] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Haibin Lin, Zhi Zhang, Yue Sun, Tong He, Jonas Mueller, R Manmatha, Mu Li, Alexander Smola, Resnest Efficientnet, and Resnext Senet. Resnest: Split-attention networks.