Serving DNNs like Clockwork: Performance Predictability from the Bottom Up

Arpan Gujarati*
Max Planck Institute for Software Systems

Reza Karimi*
Emory University

Safya Alzayat
Max Planck Institute for Software Systems

Antoine Kaufmann
Max Planck Institute for Software Systems

Ymir Vigfusson
Emory University

Jonathan Mace
Max Planck Institute for Software Systems

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. Yet the underlying execution times are not fundamentally unpredictable—on the contrary we observe that inference using Deep Neural Network (DNN) models has deterministic performance.

Here, starting with the predictable execution times of individual DNN inferences, we adopt a principled design methodology to successively build a fully distributed model serving system that achieves predictable end-to-end performance. We evaluate our implementation, Clockwork, using production trace workloads, and show that Clockwork can support thousands of models while simultaneously meeting 100 ms latency targets for 99.997% of requests. We further demonstrate that Clockwork exploits predictable execution times to achieve tight request-level service-level objectives (SLOs) as well as a high degree of request-level performance isolation.

1 Introduction

With the proliferation of machine learning (ML), model inferences are now not only commonplace but increasingly on the critical path of web requests [21,60]. Inference requests are handled by underlying model serving services [11,18,42,48] responsible for supporting scores of different pre-trained ML models (including personalized models and experimental A/B tests), ideally at low latency, high throughput, and low cost. These are demanding goals to meet at scale—Facebook alone serves over 200 *trillion* inference requests each day [39]. Furthermore, at least 100 companies are creating hardware chips for accelerated ML inference [39], which underscores the high stakes in this industry.

Yet significant software bottlenecks continue to hamper the efficient utilization of hardware accelerators, such as GPUs, for high-performance model serving. Consider an inference request passing through a model serving system. The request has an inherent deadline after which the answer ceases to be useful to the end-user, and so the system should seek to bound the latency of the request, or even provide service level objectives (SLOs) for consistently achieving low tail latency. The canonical approach for building such a low-latency system is to reduce potential wait times for resources through over-provisioning, since a larger pool of available resources makes it more likely to find a resource on which a pending request can be immediately scheduled. Increased resource provisioning, however, comes at the expense of efficiency and utilization.

Existing systems fundamentally assume that the constituent system components have *unpredictable* latency performance [11,48]. Moreover, the best-effort techniques employed to tolerate such variability, such as fair queuing, further cascade the unpredictability to other system components and propagate tail latency to higher layers. While some performance volatility of a model serving system is due to external factors, such as a bursty or skewed workload, much variability in execution times stems from design decisions internal to the service, ranging from caching decisions over conditional branching behavior to concurrency from other processes, the OS, and the hypervisor. The challenge, then, is to tame the internal unpredictability.

In this paper, we present the design and implementation of Clockwork, a distributed system for serving models with predictable performance. With an explicit focus on the ubiquitous deep neural network (DNNs) architectures we first show that DNN inference is fundamentally a deterministic sequence of mathematical operations that has a predictable execution time on a GPU. To leverage this observation in designing a responsive model serving system, we follow the principle of preserving predictability wherever possible by ***consolidating choice***: eschewing reactive and best-effort mechanisms and centralizing all resource consumption and scheduling decisions. Clockwork will only execute an inference request if it is confident the request can meet its latency SLO. To support

* Equal contribution

such proactive scheduling, Clockwork is composed of *workers* that each handle one or more GPUs, and a centralized *controller* that schedules requests. Each Clockwork worker, responsible for the exclusive model loading and inference execution on the GPUs, achieves predictable performance. If a worker cannot execute a particular schedule, because of external factors, the request is immediately aborted and the worker resumes execution of the next request at the specified time. The Clockwork controller manages the resources of each worker and maintains a minimal advance schedule for the worker's operations, including model placement and replication.

We have implemented Clockwork in C++ and evaluated it using a wide range of DNN models on production workload traces. In comparison to Clipper [11] and INFaaS [48], two prior model serving systems, Clockwork more effectively meets latency goals while providing comparable or better goodput. Clockwork more effectively shares resources between different models, and scales to thousands of models per worker. For realistic workloads comprising unpredictable, bursty, and cold-start clients, Clockwork consistently meets low-latency response times of under 100ms.

In summary, the main contributions of this paper are as follows:

- We demonstrate that predictability is a fundamental trait of DNN inference that can be exploited to build a predictable model serving system.
- We propose a system design principle, the consolidation of choice, to preserve predictable responsiveness in a larger system comprised of components with predictable performance.
- We present the design and implementation of Clockwork, a distributed model serving system that mitigates tail latency of DNN inference from the bottom up.
- We report from an experimental evaluation on Clockwork to show that the system supports thousands of models concurrently per GPU and substantially mitigates tail latency, even while supporting tight latency SLOs. Clockwork achieves close to ideal goodput even under overload, with unpredictable and bursty workloads, and with many contending users.

2 Background and Motivation

The state of machine learning. The meteoric rise of applications driven by machine learning (ML), ranging from computer vision [20, 66] to ad-targeting [3, 12] to virtual assistants [8,54], has prompted significant interest into making both ML training and inference faster. These efforts have targeted the underlying ML models, hardware accelerators, and software infrastructure. Chief among the ML modeling approaches are *deep neural networks* (DNNs), which are composed of multiple layers of artificial neurons tuned through non-linear convolution and pooling operations [17].

A plethora of specialized hardware are being developed and deployed for ML training and inference [39], such as ASIC

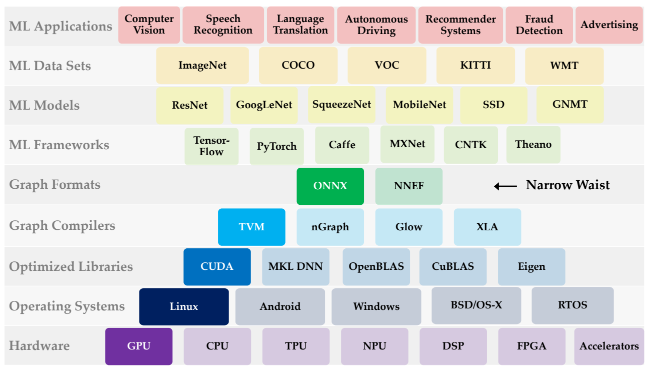


Fig. 1: Model serving targets the narrow waist of the ML software stack (adapted from Reddi *et al.* [39]). Clockwork targets the shaded blocks on the left.

and FPGA chips, Google's TPUs [34], and Facebook's Big Basin [21] chips. The dominant machine learning hardware in data centers, however, is the GPU, representing a third of the global market in 2020 [5], and will be our focus here.

Interposed between the emerging DNN applications and hardware accelerators, an ecosystem of ML software frameworks is flourishing. Fig. 1 displays several prominent projects in today's ML software stack. Layered protocol stacks in complex systems and competitive environments tend to evolve into hourglass-shaped architectures [4]. We are witnessing the ONNX and NNEF graph exchange formats for DNNs [40, 43] emerging as the "narrow waist" of the ML stack, acting as an interface between high-level ML model development and low-level software and hardware concerns.

Model serving. Operators increasingly deploy machine learning on the critical path of nascent interactive applications [60]. This has elevated machine learning inference to separate, managed *model serving* services [11, 18, 48] From the vantage point of an operator, the model serving users (customers or internal applications) upload their pre-trained DNN ahead of time (the natural format for which is ONNX/NNEF). Their applications can then submit *inference requests* to an API. The model serving back-end manages the users' models and the hardware accelerator resources, and provides timely responses to inference requests. Upon receiving an inference request, it loads the appropriate model into hardware if not already loaded, runs the DNN on the input, and returns the resulting output to the user. Model serving has similar concerns to other datacenter services [2]: it multiplexes workloads of different users concurrently and load balances requests across multiple workers and GPU hardware accelerators.

Low-latency inference. Model serving users require a timely response to their queries. Most cloud and data center services have *service-level objectives* (SLOs) that codify the performance that clients can expect from the service [33]. The most common type is a *latency SLO*, which specifies the service's acceptable request latencies, typically on the order of milliseconds [9, 24, 34]. For example, a latency SLO might specify a 10ms average response time, or a 40ms 99th percentile response time, or both. If a service fails to meet its SLOs – for example, by being too slow for too many requests – the service provider may risk a penalty.

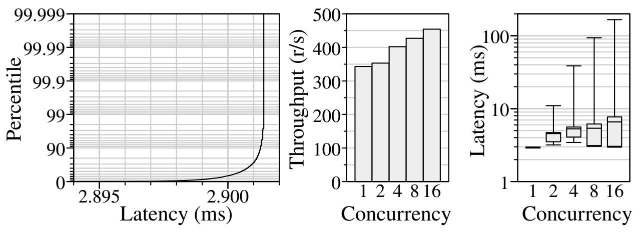
Model serving further operates under hard cost constraints. Specialized ML hardware is necessary to achieve interactive latencies [34], but it is comparatively expensive to procure and operate, and must thus be used efficiently [50, 55]. Existing model serving systems achieve efficient inferences for individual heavily used models, through dedicating entire GPUs to models and heavy use of batching [34]. However, many use cases cannot justify dedicated hardware resources: applications with insufficient request volume; specialization (*e.g.* location-specific search or language-to-language translation); and experimentation (*e.g.* retrained models and A/B testing) [53]. Efficiently serving models with low request rates requires a large number of models to share accelerators; no existing model serving system supports this.

While it is already difficult for model serving operators to meet latency SLOs under these constraints, the bigger challenge lies in minimizing *tail latency*, the insidious bane of interactive performance. Numerous sources of latency variability in complex individual [38] and distributed [13, 46] systems have been identified and studied, including out-of-order scheduling, interference from concurrency, power saving modes, and network queuing delays.

The crux of tail latency lies in performance variability of both the constituent system/network components and the encompassing architecture. To tame it, the system designer can either seek to (quoting Dean and Barrosso [13]) "*create a predictably responsive whole out of less-predictable parts*", or to expend significant effort to systematically unshroud and mitigate the performance variability of these underlying components. To meet tight tail-latency SLOs under resource constraints, the latter approach is necessary.

Observation: DNN inference is predictable. We observe that DNN executions exhibit negligible latency variability, a result both intuitive in concept — DNN inferences involve no conditional branches — and demonstrable in practice. Although we describe our observations in the context of GPU execution, they extend to other accelerators such as TPUs, and also to CPU execution where appropriate.

Conceptually, a DNN inference is a fully deterministic execution. Each DNN inference request carries a fixed-size *input* tensor argument; in practical terms this is a statically-sized array of bytes. A worker receives this input over the network into main memory. To execute on a GPU, the input is copied from main memory to GPU memory over the PCIe interconnect. The DNN is then executed on the GPU. Abstractly, a DNN is a pre-defined sequence of tensor multiplications and activation functions. Concretely, the DNN code applies these to the input tensor one-at-a-time to transform the input into an output. DNN code lacks conditional branching; input choices such



(a) CDF of 1-thread latency (b) Inference throughput and latency
Fig. 2: **Inference is predictable in isolation.** Running inferences concurrently gains up to 25% throughput (middle), at a cost of substantially increased latency variability (right).

as batching size and RNN sequence length are specified ahead of time as parameters. The output is also a statically-sized array of bytes, and it is copied from GPU memory back to main memory over the PCIe interconnect.

We compiled ResNet50v2 [66] with TVM 0.7 [10] and executed 11 million inferences using random input on the model in isolation on a state-of-the-art NVIDIA Tesla v100 GPU. We measured the latencies of each inference and show the median and high-percentile latencies in Fig. 2a. The 99.99th percentile latency was within 0.03% of the median latency.

If DNN execution times can be measured and then accurately predicted for future inferences on that model, the next question is whether a distributed model serving system can preserve the predictable responsiveness of the core inference execution.

3 Predictable Performance Principles

To build a responsive system through principled design, we further study the factors that can cause or amplify performance variability. Importantly, components at any level of the modern system stack can contribute to variable request latency, whether at the application layer, in the operating system, or even in the hardware [38]. Network effects and workload fluctuations add two more sources of unpredictability to distributed systems.

The whole is more than the sum of its parts. The overall system performance variability is primarily governed by how it is assembled of its constituent components. We can handle variable latency of a software component in several ways. First, we can ignore the problem and allow the volatility to propagate to later requests or percolate to other components of the system. Even performance-conscious code that is optimized to improve throughput or average latency does not fix tail latency [14]. An example of this contagiousness of unpredictability, known as the "straggler" problem in data analytics frameworks [6, 46], is when a worker executes a request that takes unusually long and the other requests that were enqueued on the worker in the meantime and then incur the extra delay from the unexpected wait-time. Ignoring the variability can further compound the problem across the system, such as when the handler itself has variable latency [58].

Second, we can mitigate the volatility by delaying the request until the worst-case latency, thus exchanging lower resource utilization for predictability—often a steep price when worst-case latency is significantly higher than the median.

Third, we can minimize variability by expending more resources, again in trade for lower utilization. Some networked systems, for instance, are designed to submit the same job to multiple workers in parallel and then to cancel the jobs upon successfully receiving a result from fastest worker [13].

Fourth, upon detecting an unusual delay, we can notify a throttling or feedback mechanism and lower the impact on future requests. Such "best-effort" methods are typically reactive and aimed at longer-term effects, such as by adding more resources (auto-scaling [15]) or by balancing load.

Consolidating choice. We take a fundamentally different approach: *designing a predictable system from the bottom up.* Our strategy is to eliminate choices from lower system layers—a philosophy based on our observation that when executing an essentially predictable task, performance variability only arose when a lower layer in the system was given choices in how it to execute its task. Examples from all layers of the systems stack abound, including:

- **Hardware level:** when a GPU is passed multiple CUDA kernels to execute in parallel, the GPU has the choice of how to allocate resources, including execution units and memory bandwidth, between kernels. The GPU makes these choices based on its internal state and undocumented, proprietary policies.
- **OS level**: when we create multiple threads that the operating system can execute on the same core, the OS has the choice of what threads to execute when, based on internal scheduling policies and state.
- **Application level**: when the worker processes of a distributed application each manage their own cache independently, the workers have the choice of what to cache and for how long, leading to unpredictable hit rates and latency variability [31]; similarly, when worker processes implement their own queuing policy, they have the choice of which requests to execute first, leading to unpredictable queuing times.

Fig. 2b illustrates this: a standard design for building a worker would use thread pools serving inference requests in parallel to saturate the GPU. While concurrent threads indeed increase inference throughput by up to 25%, the factors above cause tail latency to increase by 100×.

Our design principle is to *consolidate choices* in the upper layers: once a layer implements choices for lower layers based on internal state, it forces the lower layer to follow a narrow path of possible executions, causing the performance of the resulting layer to be nearly deterministic. The upper layer can then predict the performance of the lower layers and reason with foresight about resource utilization and the anticipated execution times for all requests. The price of this strategy, however, is a tighter coupling of components and a less modular architecture.

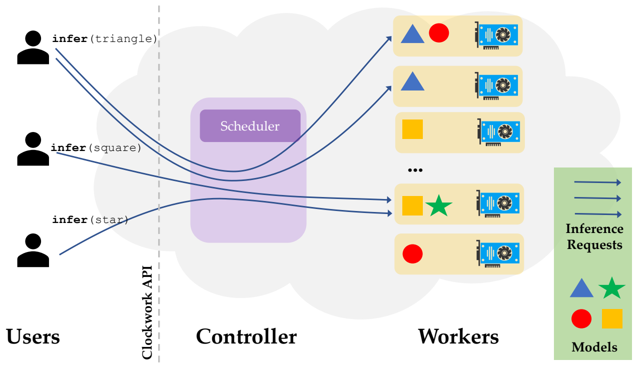


Fig. 3: The Clockwork architecture.

4 Design

By recursively applying the principle of eliminating choice from lower layers, we converge on a design where all performance-critical execution choices are made in the topmost layer. In the context of a model serving service, this process converges to an architecture, which we call Clockwork, with a centralized controller and workers with predictable performance.

4.1 Overview

Architecture. The centralized Clockwork controller accepts and schedules incoming inference requests and then delegates them to the workers for predictable execution (Fig. 3). Each worker has a set of DNN models loaded into main memory and maintains exclusive control over one or more GPUs. By pushing all execution choices to the controller, the scheduler in the controller has a global view of the system state including all workers. At any time it can make accurate, high-quality caching, scheduling, and load balancing decisions. The controller can perform these actions proactively the moment it predicts a problem, because the workers are predictable. The controller transmits continual scheduling information to the workers that, by design, will execute schedules exactly as directed.

Illustrative example. To elucidate the Clockwork architectural components with more detail, including the choices that were consigned to the controller, consider the keys steps involved to serve an inference API call against a hypothetical model called `star` (Fig. 4).

Upon receiving an inference request with a 30ms SLO, the controller is aware that a target worker has yet to copy the model weights (denoted by ★) from RAM to GPU memory. It calculates the anticipated time for copying the model weights and input vector, plus the time required to execute the inference, and concludes that the request will complete within the specified SLO. Consequently, the controller instructs the worker to copy the model weights to the GPU via a LOAD action (a *cold-start*) followed by an INFER action. Dotted lines show when the GPU is busy. Since the controller is aware of all timings, it needs no acknowledgement from the worker until the DNN output is ready (or the worker unexpectedly

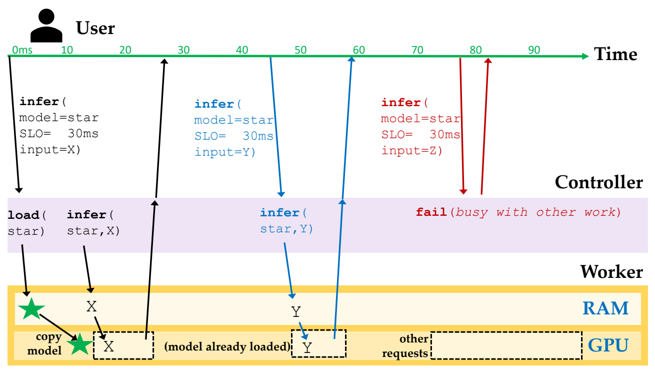


Fig. 4: Timeline of three illustrative inference requests.

cannot execute the actions as directed). Last, the result is returned back to the user. Each step of this execution is fast; for ResNet50, LOAD and INFER take approximately 8 ms and 3 ms respectively (see Appendix A for a detailed breakdown).

Next, the user submits a second request (blue) for the `star` model. Since the worker has just loaded the model's weights into memory, the controller instructs the worker to conduct an INFER action. Note that the round trip time for the second request is lower than for the first request, and both requests meet their SLOs.

Finally, the user issues a third request (red), again to the same model. The controller is aware that the model weights are present in the target worker's GPU memory. However, the controller has prioritized other requests to execute on the worker's GPU. Knowing this request cannot meet its target SLO, the controller does not proceed to schedule an INFER action. The system cancels the request before performing any fruitless work.

4.2 Consolidating Choice

We apply the principle to our design in three main ways. First, changes in the worker's state, for instance evicting a DNN from GPU memory, can influence performance for future requests in a way that makes performance estimation complex. We therefore maintain an invariant that no worker operation can have implicit performance side-effects on any future operation. Second, we must ensure that a predictable component either delegates scheduling decisions that may impact performance to the centralized controller, or otherwise makes schedules deterministic. Third, when a predictable component is unable to execute a schedule as instructed, it is treated as an error to enable workers to get back on schedule. Workers do not attempt best-effort remediation, so as to avoid cascading mispredictions.

We enforce these properties in Clockwork through an action command abstraction between the controller and workers that, in lieu of traditional RPC calls, either communicates a change in a worker's state or a task for a worker to execute. Each action the controller issues to a worker, such as LOAD and INFER, has predicted execution time and a designated execution window. These are derived using the known state of the worker, previously submitted actions, and known transitions in controller-maintained worker state.

4.3 Challenges for Predictable Inference

To fully consolidate choice and achieve predictable execution, we must first identify where performance-critical choices arise in system components. We have established that DNN inference itself on a GPU has deterministic performance; we next study the challenges in extending this result to a full-fledged inference system.

Managed memory and caches can be unpredictable (C1). RAM and GPU memory on a worker constitute state that impacts the performance of future requests. Additionally, some memory allocators exhibit variable timing for allocation and deallocation requests due to internal trade-offs between memory fragmentation and amortized performance. Memory that is used as a cache specifically introduces performance variability between cache hits and misses, with an internal cache replacement policy influencing performance of future items. To maintain predictability we must instead consolidate choice by managing cache admission and eviction for each worker at the central controller. Fortunately, DNN weights caching is coarse-grained and per-model.

Hardware interactions can be unpredictable (C2). Many system resources are implicitly administered by hardware schedulers that operate at very fine time-scales and produce different schedules under even minute shifts in the arrival times of other requests. The volatility of timing coupled with proprietary and un-documented scheduling policies make it difficult to accurately predict completion times for concurrent requests. The remedy for non-determinism is to strip away the ability for schedulers to reorder requests by forcing only a single request to be executed at a time, at the cost of spending greater effort on keeping the resource fully utilized. Fortunately for DNN inference, one-at-a-time execution on GPUs does not sacrifice much throughput compared to concurrent execution (Fig. 2b).

External factors can trigger performance variance (C3). Even after systematically removing the key internal sources of unpredictability by consolidating choice, there will always remain external sources outside of the controller's purview. These include performance interference through shared network bottlenecks, thermal throttling of CPUs and GPUs, and others. The only option is to minimize their effects by building sufficient tolerance into the system.

4.4 Predictable DNN Worker

At a high-level, Clockwork workers maintain DNNs in memory and execute inference requests on one or more GPUs. The workers interface with the controller to receive actions.

Memory management. Model weights must be present in GPU memory to execute an inference. However, GPU memory capacity is small (≤32GB) relative to host memory (≤4TB), and host-to-GPU memory transfers (≈8.3ms for ResNet50) typically take longer than running the DNN inference on the GPU (≈2.9 ms). Consequently, Clockwork treats GPU memory as a cache, letting commonly or recently used models avoid expensive loads. To overcome **C1**, workers explicitly expose LOAD and UNLOAD actions to the controller for copying models to and removing models from worker's GPU memory with deterministic latency. These actions also update the state that the controller tracks for the worker.

Inference execution. The controller only sends an INFER action when a model is present in GPU memory or a LOAD action will momentarily complete. The worker internally divides INFER actions into three steps. First, INPUT transfers the input vector from host to GPU memory. Next, EXEC performs the actual heavy-weight DNN GPU calculations, which dominate the total inference time. Finally, OUTPUT transfers the resulting output vector from the GPU back to host memory. These steps may coincide: the previous request's outputs can be copied at the same time as the current request's input is being transferred. However, multiple concurrent EXEC calls cause the GPU hardware scheduler to behave unpredictably (**C2**). Fortunately, a DNN inference call by itself is highly parallel and fully occupies the GPU while also restricting the hardware scheduler to a single, predictable option (Fig. 2b). Clockwork workers therefore run a single EXEC at a time, a design choice that reduces performance variability by two orders of magnitude while only minimally decreasing inference throughput (Fig. 2b).

Interface with the controller. Clockwork workers receive LOAD, UNLOAD, and INFER actions from the controller with detailed timing expectations attached:

`type`	INFER, LOAD, or UNLOAD
`earliest`	the time when this action may begin executing
`latest`	when this action will be rejected

Rather than executing actions in a work-conserving, best-effort manner, workers strictly follow the schedule of actions imposed by the controller. The controller communicates two timestamps with every action, `earliest` and `latest`, to designate a time interval during which the worker may begin executing the action. Actions that cannot start within the prescribed window are cancelled and never executed. This allows workers to quickly get back on schedule after an individual action is delayed unexpectedly (**C3**) by skipping one or more actions, minimizing the impact the delay on other actions. Workers also communicate the result of each action back to the controller, including whether the command was successful and the measured execution time.

4.5 Central Controller

All decision-making in Clockwork occurs in the central controller. The controller receives inference requests from users, and translates them into worker actions while striving to meet SLOs.

Modeling worker performance. The controller maintains a per-worker performance profile based on telemetry information and updates it regularly to tolerate shifts due to external factors (**C3**). The controller also tracks the outstanding actions and memory state at every worker. Since actions have inherently deterministic latency by design, the controller can deduce the earliest time that a worker could begin executing a new action.

Action scheduler. Combined, a global view of system requests, up-to-date worker performance profiles, and a mechanism for accurately predicting when outstanding actions will complete allow the Clockwork controller to proactively manage action schedules for workers. The controller attempts to pack these worker schedules tightly by making narrow, realistic estimates for the `earliest` and `latest` time interval. The interval width balances a trade-off between Clockwork SLO fulfillment and system goodput. On one hand, making the interval too narrow increases the risk of an action not being executed by a worker because it could not be completed in time (**C3**), potentially triggering an SLO violation. On the other hand, underestimating the window length can create periods of inactivity and decrease worker utilization, thus affecting Clockwork goodput.

The scheduler lazily decides which worker should execute the inference. The controller only submits a minimal amount of work to keep workers utilized; it is in no hurry to commit because it can accurately prediction action timings. Delaying choices on the controller improves schedules by providing more options, permitting the Clockwork controller to re-order and *batch* inference requests to the same model, significantly improving resource efficiency and throughput.

In our design, any worker can process any request since they all store every model in host memory; however, workers have different sets of models loaded into their GPU memory. A worker that executes only cold inferences must transfer weights for each model from host memory to the GPU and may saturate the available PCIe bandwidth, whereas a worker that executes only hot inferences may be bottlenecked by the GPU. The Clockwork scheduler balances load by mixing and matching hot and cold inferences among all workers.

5 Implementation

Clockwork's implementation, comprising 26KLOC of C++, contains various decisions that enable Clockwork to consolidate choice on its controller.

5.1 Models

Predictable model execution. Prior model serving systems such as Clipper [11] and INFaaS [48] act as orchestration layers atop existing model execution frameworks such as TensorFlow [1] and TensorRT [41]. This decoupling makes it difficult to consolidate choice, since the model execution frameworks encapsulate scheduling and memory management decisions that we wish to make with Clockwork. Instead, Clockwork implements its own model runtime, reusing key components of the TVM optimizing compiler [10]. Clockwork's model runtime enables fine-grained control over each stage of a model's execution. For models provided to Clockwork (*e.g.* in ONNX form), we compile a binary representation using TVM and postprocess the model to produce the following:

- **Weights:** A model's weights are a binary blob (10s to 100s of MB, cf. Appendix A).
- **Kernels:** The CUDA kernels that execute a model (10s to 100s of kB). These are not provided by the user; they are derived from the abstract model definition, and kernels from different users can safely execute within the same process. Clockwork uses the kernels compiled by TVM. Clockwork compiles kernels for multiple batch sizes; by default 1,2,4,8, and 16. Kernels for different batch sizes can use the same weights without modification.
- **Memory metadata:** At runtime, models do not directly allocate memory; instead, Clockwork pre-allocates any memory needed and passes pointers as arguments to function calls. The memory requirements for a model are static, and Clockwork precalculates the required workspace memory and offsets required for each kernel.
- **Profiling data:** Clockwork runs a brief profiling step to produce a seed estimate for model execution times (cf. Appendix A).

Model loading. Models are stored in an efficient serialized form on disk. Clockwork workers pre-load models from disk into main memory on worker startup. For the worker machines used in our evaluation, 768GB RAM can support thousands of models (cf. §6.5). Once a model is in main memory, Clockwork extracts and links the CUDA modules needed for its execution. To improve predictability, Clockwork disables JIT compilation and caching of CUDA kernels. Although not evaluated here, Clockwork also supports dynamic model loading over the network, where loading and unloading typically take less than 1ms.

5.2 DNN Workers

Each machine runs one worker process, that receives and executes actions from Clockwork's controller. We do not run Clockwork in a container or VM, to avoid the performance interference that sharing can introduce.

Managing model weights in memory. Clockwork pre-allocates all GPU memory and divides it into three categories:

- **Workspace:** Models require a variable amount of GPU memory for intermediate results. This memory is transient and only needed during execution; once an output has been produced, it is no longer needed. Clockwork only executes models one-at-a-time, so it allocates 512MB workspace memory.
- **IOCache:** Although Clockwork only executes models one-at-a-time, Clockwork asynchronously copies inputs to the GPU prior to execution, and outputs to host memory after execution. Clockwork allocates 512MB device memory for temporary storage of inputs and outputs before and after execution.
- **PageCache:** The remaining device memory is used for storing model weights, divided into 16MB *pages*.

Clockwork's PageCache has several advantages. First, avoiding repeated memory allocation calls leads to more predictable executions, since memory allocation can be an unpredictable source of overheads (**C1**). Second, paging *simplifies choice*: external memory fragmentation issues are eliminated, and the controller need only track the number of total free pages to completely capture the worker's memory state. Paging slightly increases memory utilization; however, model memory requirements are static and known ahead of time, and can be bucketed on to pages to reduce internal fragmentation. Paging does not affect the latency of memory transfers.

Actions. To orchestrate workers, the controller uses the previously described *action* abstraction. Actions contain a unique `id` and an action-dependent `payload` (*e.g.* INFER inputs). Each worker runs a dedicated *executor* for each action type and each worker-GPU. An executor runs a thread that dequeues actions chronologically by `earliest` timestamp, and waits until `earliest` is reached before proceeding with an action. Executors reject actions whose `latest` timestamp has passed. To reduce interference between threads and other processes, each executor is pinned to a dedicated core and runs at real-time priority. Both INFER and LOAD execute asynchronous work in their own CUDA streams. Each executor is bottlenecked by a different resource (*e.g.* GPU execution and PCIe transfers) and can run concurrently with negligible interference.

Results. A network thread maintains a persistent connection with the controller for receiving actions and sending results. A result comprises the following:

`status`	success or an error code
`timing`	start and end times, and on-device execution duration for any asynchronous work

LOAD actions acquire pages from the PageCache, then copy weights to those pages. If no pages are available then LOAD aborts. The controller explicitly frees pages with UNLOAD; this only updates in-memory metadata and always succeeds.

INFER actions comprise INPUT, EXEC and OUTPUT, each of which have dedicated executors. INPUT executes immediately on receipt of INFER; it acquires IO memory from the IOCache then copies inputs. EXEC inherits the INFER action's `earliest` and `latest` timestamps; it checks weights and inputs are present then executes kernels on the GPU, using Workspace for intermediate calculations. OUTPUT immediately copies outputs back to main memory then releases the IO memory. To simplify controller decision making, INPUT and OUTPUT are not exposed as actions since they are orders of magnitude faster than EXEC and LOAD (10s of microseconds) for our workloads. Clockwork's memory management allows for back-to-back INFER actions for the same model.

5.3 Central Controller

Clockwork's centralized controller is responsible for all decision making. On startup, it establishes persistent connections

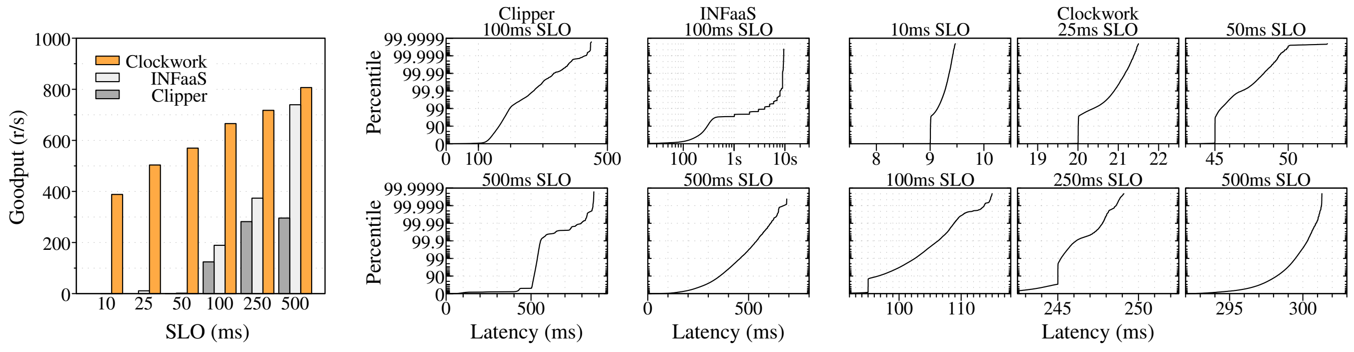


Fig. 5: Throughput and latency measurements for Clipper, INFaaS, and Clockwork. We deploy 15 instances of ResNet50 on 1 worker; each model submits 16 concurrent requests in a closed loop. (Left) Request goodput. Goodput only counts requests that succeed within the SLO. (Right) Request latency CDFs across all requests (including those rejected due to missed deadlines). Latency CDFs are scaled to highlight tail latency.

to all workers and exchanges metadata about the size of each worker's PageCache, the models present on each worker, and their initial profiled execution times. The core duty of the controller is to satisfy requests received from clients by submitting actions to workers. This decision making is encapsulated in the *Scheduler* interface:

`onRequest`	client request received, specifying a model ID, SLO, and providing inference inputs
`onResult`	a result is received from a worker

A scheduler implements this interface, and can invoke `sendAction` to send an action to a worker, and `sendResponse` to respond to a client. A separate layer of the controller implements common tasks such as networking, forwarding inputs to workers, setting timestamps, and handling timeouts. This design concentrates all choice in a single place, and enables different scheduler implementations to be easily dropped in.

Managing worker state. The scheduler makes decisions across all workers, and relies upon predictable executions on the workers. Key to leveraging worker predictability is to maintain an accurate representation of the worker's execution state, which is threefold: *memory state*, in which the scheduler tracks what models are present in the worker PageCache and when LOAD will be required; *action profiles*, which are measurements of past 10 actions duration, stratified by action type, model and batch size, to predict the duration of future action; and *pending actions*, which tracks submitted actions and estimates when each executor will next be available. Taken together, these enable the scheduler to accurately predict when candidate actions will complete, and avoid submitting work that cannot complete before the request's deadline.

Scheduler. Clockwork's scheduler is conceptually simple because all choice is concentrated in a single place. Instead of eagerly sending actions to workers, it ensures only a minimal amount of work is outstanding on each worker at any point in time. By default, Clockwork schedules 5 ms into the future, and schedules new actions only when a worker drops below this threshold. This is only effective due to highly predictable worker executions; inaccurate estimates would lead to underutilized workers, or require scheduling more work in advance. Clockwork schedules INFER actions from a single global request queue, prioritizing batching where possible. Clockwork schedules LOAD actions by estimating each model's SLO violations, given the model's current state and outstanding requests, and selecting the model with the most estimated violations. A detailed description of Clockwork's scheduler is provided in Appendix B.

6 Evaluation

In this section we evaluate Clockwork's ability to reliably serve DNNs under a variety of workload conditions. We begin our evaluation with simple workloads in controlled settings, before expanding to heterogeneous models and diverse workloads. Our evaluation shows that Clockwork's assumptions about predictability hold, and result in a system that can effectively meet SLOs and drastically reduce tail latency.

Experimental setup. We deploy Clockwork in a private cluster of 12 Dell PowerEdge R740 Servers. Each server has 32 cores, 768 GB RAM, and 2×NVIDIA Tesla v100 GPUS with 32 GB memory. The servers are connected by 2×10 Gbps Ethernet on a shared network. In all experiments, we run the controller, clients, and workers on separate machines.

6.1 How Does Clockwork Compare?

We begin with a comparison to two prior model serving systems, Clipper [11] and INFaaS [48]. For Clipper and Clockwork, we provision a single cluster machine to use 1 GPU to serve 15 separate copies of ResNet50. ResNet50 is the *de facto* model used for comparison previously by these systems; we chose 15 models as this reached the memory limit of Clipper[1]. To evaluate INFaaS, we deployed an `m5.24xlarge` and an `p3.2xlarge` EC2 instance as the master and the worker, respectively. These are not identical experiment conditions; however, INFaaS is tightly integrated with EC2, and it was not

[1]INFaaS memory limits were reached at 64 models

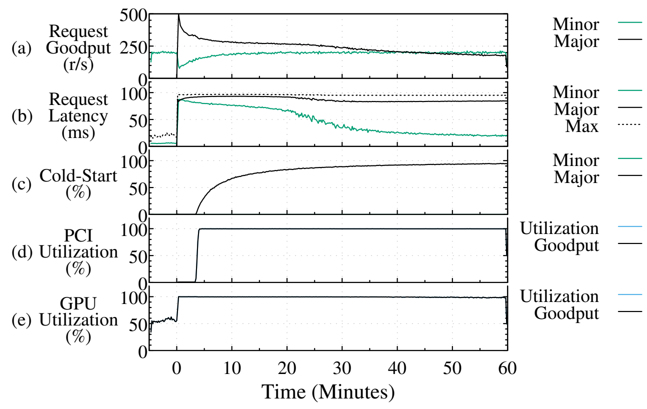


Fig. 6: Clockwork can serve thousands of models from a single worker. From $t = 0$, the Major workload adds an additional model per second, to a total of 3,600 models at $t = 60$ (cf. §6.2.)

possible to deploy it on our cluster infrastructure. We include these results for qualitative comparison.

For each model, we run 16 closed-loop clients[2]. Requests to the same model can be batched, but requests among different models are not batched. We run multiple experiments, varying the target SLO from 10 ms to 500 ms. Fig. 5 plots the *goodput* achieved by each system as the target SLO varies from 10 ms to 500 ms. Goodput is the number of requests returned within the target SLO; an ideal system that meets the deadlines of all admitted requests attains 100% goodput.

With a high SLO of 500 ms, Clockwork and INFaaS meet their SLOs and have comparable throughput of approximately 800 r/s. Clipper's goodput is substantially lower, as Clipper only treats SLOs as an average latency target, not a strict threshold, and converges to this target over time without bounding latency variability. As SLOs tighten, goodput and tail latency deteriorate for both Clipper and INFaaS, and their goodput collapses below a 100 ms SLO. Like Clipper, INFaaS uses the SLO as a coarse-grained goal for reactive policies. Consequently, only Clockwork can continue serving SLOs below 100 ms.

Fig. 5 also plots latency CDFs for Clipper, INFaaS, and Clockwork. We scale the CDFs to emphasize tail latency. The figure illustrates how both Clipper and INFaaS allow latency higher than their SLOs. However, of note, with a 500 ms SLO, INFaaS successfully finds a configuration that can serve this SLO, and meets its SLO for 99% of its requests. By comparison, Clockwork's tail latency remains very close to the SLO in all cases. For the 500 ms SLO, Clockwork's latency remains at ≈300 ms because it schedules each model's entire batch of 16 requests at a time, round-robin across models. With 15 models and a 20 ms batch-16 execution duration, Clockwork does not exceed the optimal 300 ms latency.

6.2 Does Clockwork Scale Up?

The previous experiment represented an idealized scenario, with only a small number of models, each with a steady sustained workload. We now examine the serving limits of a single worker. We deploy 3,601 copies of ResNet50 to a worker, and set a 100 ms SLO. We submit two workloads: a Major workload and a Minor workload. The Major workload comprises 3,600 model instances; we vary the number of instances that are active at any point in time, and evenly distribute a workload of 1000 r/s across all active models. The Minor workload is a single model instance that maintains a fixed 200 r/s request rate throughout the experiment.

Figure Fig. 6 (a) plots the total goodput achieved throughout the experiment. From t=−15 to t=0 (we denote t in minutes) only the Minor workload is present, achieving its full 200 r/s. At t=0, we activate one model instance of the Major workload; the addition of 1000 r/s fully saturates the GPU (e). After that, we activate an additional model of the Major workload every 1 second. As more model instances become active, the Major workload's goodput drops since each additional model forgoes batching opportunities. At t=60 all 3,600 models are active, each submitting approximately 0.28 r/s.

By t=3.5, 201 models have been activated, reaching the capacity of GPU device memory. To continue serving requests, Clockwork begins swapping models on and off GPU; Fig. 6 (d) shows PCI utilization rapidly rises to 100%. As more models activate, an increasing number of requests in the Major workload find that their model is not loaded; Fig. 6 (c) plots the rise in *cold-starts* until it is nearly 100% by the end of the experiment. The minor workload, with its sustained request rate of 200 r/s, does not experience any cold starts because its demand dwarfs every other model after the first 5 seconds. As the number of cold-starts increases, the demand on GPU execution decreases, enabling the Minor workload's goodput to gradually grow back to 200 r/s. At approximately t=20, the bottleneck for the Major workload shifts to PCI utilization, enabling the Minor workload's latency to drop back to an average of 20 ms (b).

This experiment illustrates how bottlenecks in Clockwork can shift as workload demand changes. Clockwork can deal with shifting bottlenecks even while serving a large number of models. As illustrated in Fig. 6 (b), the maximum request latency across the experiment did not exceed the 100 ms SLO. The overall resource goodput, shown in Fig. 6 (d) and (e), was always equal to the total utilization.

6.3 How Low Can Clockwork Go?

Clockwork's predictability and centralized decision-making enables it to satisfy low-latency SLOs. In this experiment, we use 6 Clockwork workers and evaluate the lower limit on SLOs that Clockwork can achieve, by varying the SLO and measuring the proportion of successful requests. We repeat the experiment for six different workloads, varying the number of ResNet50 instances ($N = 12$ or 48) and cumulative request rate ($R = 600$ r/s, 1200 r/s, or 2400 r/s). For each experiment run, we begin with an SLO of 2.9 ms (1× the execution latency of batch-1 ResNet50 inference). Every 30 seconds we increase the SLO by 50%; by the end of the experiment the SLO reaches 250ms.

[2]Open-loop clients yielded similar results

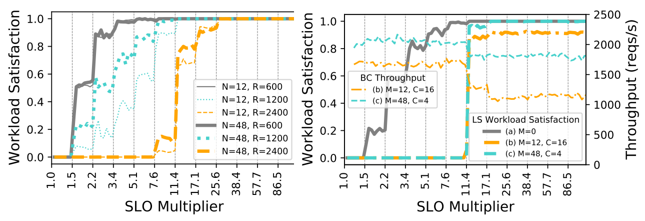


Fig. 7: (Left) Workload satisfaction rates for latency-sensitive (LS) requests with varying SLOs. Request rates are provided in requests per second. (Right) Workload satisfaction rates for LS requests and throughput for batch client (BC) requests when the LS requests SLOs are varied.. In both figures, the vertical lines show SLO changeover times and are labeled with SLO multipliers.

We run a separate open-loop client for each model with a Poisson inter-arrival time distribution, and as before, all models are independent (requests cannot be batched across models).

Fig. 7 (left) plots the *workload satisfaction* for each experiment run, *i.e.* the proportion of requests whose SLO was satisfied. For a load of R=600 r/s and 1200 r/s, irrespective of the number of models, Clockwork successfully satisfied tight relative SLOs of 3.4 and 7.6 (10 and 22 ms), respectively. Even at R=2400 r/s, Clockwork comfortably managed an SLO of 25.6 (≈75 ms).

6.4 Can Clockwork Isolate Performance?

Clockwork can satisfy tight SLOs for latency-sensitive clients in isolation; we next consider when the system is shared with other users serving batch requests without latency SLOs. We use As before, we use 6 Clockwork workers. We provision multiple ResNet50 instances and divide them into two separate categories: latency sensitive clients (LS) and batch clients (BC). There are $N = 6$ LS instances each submitting 200 r/s in an open loop. BC clients generate requests in a closed loop, thereby ensuring Clockwork is saturated with requests at all times. We vary M, the number of BC clients, as well as C, the number of concurrent requests each BC client can submit; varying C affects the maximum achievable batch size for a BC client. We considered three different scenarios: **(a)** baseline without batch clients ($M = 0$); **(b)** few ($M = 12$) batch clients with high concurrency ($C = 16$) thereby permitting large batches; and **(c)** many ($M = 48$) batch clients with low concurrency ($C = 4$), thereby permitting only small batches.

Fig. 7 (right) illustrates the workload satisfaction rates for LS clients and the total throughput achieved for BC clients. Clockwork can successfully prioritize LS requests over BC requests. Through SLO-aware scheduling, it ensures that the workload satisfaction rates for LS requests are not affected by the presence of other pending, less time-critical requests. At the same time, Clockwork does not throttle BC requests entirely, but schedules them when idle time is available or expected in the immediate future. However, when the SLOs for LS requests are too tight (SLO multiplier lower than 11.4), many LS requests are rejected in advance, allowing pending BC requests to pass through. Execution of BC requests in large batches (scenario (c)) prevents LS clients from achieving 100% workload satisfaction rates at high request rates, regardless of the SLO.

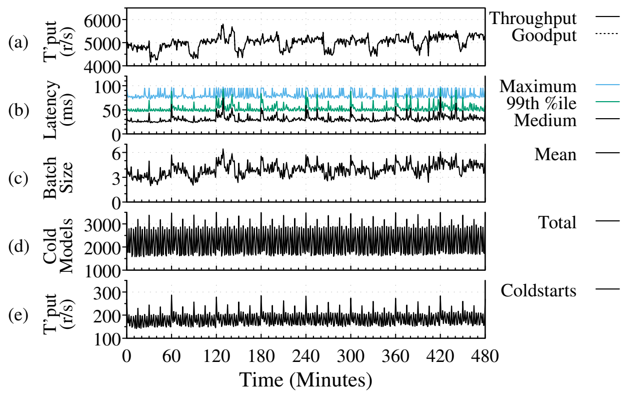


Fig. 8: Microsoft Azure Functions (MAF) over Clockwork; see §6.5 for a description.

6.5 Is Clockwork Predictable?

We now ask whether executions remain predictable under realistic workloads that comprise many concurrent users and models. We also investigate whether Clockwork effectively exploits this predictability.

To answer these questions, we deploy Clockwork on 6 workers and replay a workload trace of Microsoft Azure Functions (MAF) [51]. The trace records approximately 17,000 function workloads, counting the number of invocations of each function, every minute, for 2 weeks. It interleaves a wide range of workloads, including heavy sustained workloads, low utilization cold workloads, bursty workloads that fluctuate over time, and workloads with periodic spikes [51]. We believe this to be a representative workload for evaluation since serverless platforms enable a wide range of applications and supporting ML inference on serverless is an active area of research [7, 32].

In this experiment, we replay 8 hours of the MAF trace in real-time. We use 61 different model varieties taken from the Gluon model zoo [20] (cf. Appendix A) and duplicate each 66 times, resulting in a total of 4,026 instances[3]. We replay four or five function workloads for each model instance. We configure Clockwork with a 100 ms SLO.

Clockwork with realistic workloads. The time series in Fig. 8 (a) shows the throughput and goodput achieved across all models. For the 8 hour experiment, the average workload throughput was 4,860.6 r/s, and Clockwork provided 4,860.5 r/s of goodput. Out of a total of 140 million requests, 4,511 failed due to action timing mispredictions and timed out at 100 ms. No request exceeded 100ms. Fig. 8 (b) plots the median, 99th percentile, and maximum request latency over the course of the experiment. Periodic latency spikes occur every 60 minutes (and to a lesser extent every 15 minutes) due to the presence of periodic hourly workloads [51]. During these spikes, the overall request latency increases. Workload spikes

[3]4,026 models reaches the main-memory capacity of our worker machines

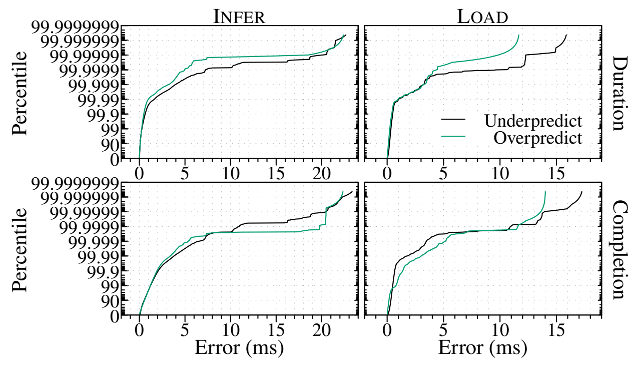


Fig. 9: Clockwork prediction and completion errors for MAF trace.

do not cause SLO violations because of latency headroom; Fig. 8 (c) shows the average batch size for the experiment, and with each workload spike, Clockwork can schedule larger batches, with higher latency. To evaluate the cold-start behavior of this workload, we categorize a request as a cold-start if its model is not already loaded into GPU's memory before arrival. For each 1-minute interval, Fig. 8 (d) counts the number of unique models that have at least one cold-start. On average, 2190 unique models perform cold-starts each minute; or approximately 54% of all models. However, while many models perform cold-starts, they only represent a small fraction of all requests. Fig. 8 (e) plots the throughput of cold-start requests, averaging 178 r/s, or 3.7% of all requests. These results show that Clockwork can sustain significant load for varied, realistic workloads comprising thousands of models.

Predictable executions. Since Clockwork's scheduler relies on accurate predictions of action latency, and to evaluate Clockwork's underlying assumptions of predictability, we now evaluate the accuracy of Clockwork's predictions. We measure the latency of INFER and LOAD actions on Clockwork's workers and compare it to the time estimated by Clockwork's controller to derive a *prediction error*. Prediction errors comprise two types: *overprediction*, when the real execution latency is faster than predicted; and *underprediction*, when the real execution latency is slower than predicted. Consistent overpredictions can lead to idle resources, while consistent underpredictions can cause SLO violations. Fig. 9 (top) plots the prediction errors for INFER and LOAD actions. For INFER actions, the 99th percentile of both overpredictions and underpredictions is 250 μs. Thereafter, the tail latency grows to more than 20 ms in a few extremely rare cases. Clockwork consistently underpredicts more than it overpredicts, as it uses a rolling 99th percentile measurement to make its predictions. For LOAD actions, the 99th percentile of overpredictions and underpredictions is 338 μs and 240 μs, respectively.

Fig. 9 (bottom) plots the *completion time error*. Clockwork must accurately predict when a given action will complete, taking into account any previously submitted actions. Individual prediction errors can compound, leading to increased completion time error. For INFER actions, the error compounds 4×, with a 99th percentile completion error of ≈1 ms. In extreme cases, Clockwork's completion error also grows to more than 20 ms. However, the completion error does not substantially exceed the action duration error, implying that for Clockwork, erroneous predictions of outliers are statistically independent. All extreme mispredictions were one-off occurrences.

Tighter SLOs at larger scale. To explore Clockwork's scalability, we performed a final experiment using the MAF workload traces. We deployed 10 Clockwork workers, each utilizing 2 GPUs, and replayed the workload trace scaled up 1.5× for one hour. We repeated the experiment twice, once with an SLO of 100 ms, and once with an SLO of 25 ms. The experiment exceeds the network capacity in our testbed since Clockwork routes all inputs via the controller (this is not fundamental; see §7 for discussion). To bypass this limitation and examine Clockwork's scalability, we modified our clients to send zero-length inputs and have workers randomly generate inputs when INFER actions arrive.

SLO	Goodput	Missed SLO	P50	P99.99
100ms	6174r/s	0	6.28ms	49.92ms
25ms	6060r/s	361	5.77ms	21.60ms

The average request rate for the experiment was 6,174 r/s. With a 100 ms SLO, Clockwork schedules all requests well within the target. At 25 ms SLO, 1.8% of requests time out without executing, while a total of 361 requests (0.00002%) were admitted, took longer than expected, and exceeded the 25 ms SLO. The maximum request latency was 43.7 ms. This experiment demonstrates that even very strict SLOs can be achieved by Clockwork workers.

6.6 Summary

In comparison with prior model serving systems, Clockwork achieves superior goodput, can serve substantially more models concurrently, and violates substantially fewer SLOs. Due to a lack of performance variability, Clockwork can achieve much tighter latency SLOs without sacrificing tail latency. Clockwork's underlying assumptions about predictable executions bear out in reality: by consolidating choice it is possible to build a predictable system that substantially curtails tail latency.

Clockwork extends to a diverse range of workload conditions not supported by prior systems, including supporting thousands of models concurrently per GPU. Slow cold starts can run alongside high-throughput workloads without interference. Under all workload conditions, including cold starts and even under overload, Clockwork meets most SLOs without degrading service, and maintains close to maximal possible goodput. Finally, Clockwork isolates users of different models, enabling low-latency workloads to share the same system with background batch workloads.

7 Discussion

Machine learning. Clockwork focuses on DNN inference, and excludes data preprocessing and postprocessing steps that are user-defined and CPU-bound. Safely and predictably executing these in Clockwork is a current research topic. Expanding Clockwork into other ML paradigms, such as deep reinforcement learning, DNN training, and composite applications, like model pipelines and cascades, raises philosophical questions about the nature and limits of predictability.

Inference accelerators. The Clockwork approach generalizes readily beyond GPUs to other inference-specific hardware accelerators [39], whose performance is arguably even more predictable. TPUs, for instance, are explicitly built around the idea of delegating control to software.

Network. Clockwork does not explicitly consider the network in its scheduling decisions; the occasional network latency spikes of dozens of ms during our experiments had negligible impact on our results. Our prototype routes all inputs and outputs through the central controller which will become a bottleneck at scale.We were able to reach the limits of our testbed network with six workers and a sustained, single-model workload. However, Clockwork's controller only requires request metadata to schedule requests, and we are working to remove this limitation with a tier of load balancers. Centralized scheduling is itself a scalability bottleneck, but prior work demonstrates that centralized schedulers can nonetheless reach impressive scale [16, 47].

Security. Security is important for all multi-user systems, since there are no container or hypervisor boundaries separating the workloads of different users. Clockwork does not explicitly address security; however, Clockwork does not execute arbitrary user code. Users must submit models in an abstract format that we then compile to binary code under the covers. Clockwork's threat model resembles shared storage or database systems, where system correctness is the chief concern; we have not verified any safety properties of Clockwork.

Fault tolerance. While Clockwork is a distributed system, we do not address the challenges of tolerating failures when serving models at large scale. This will require implementing a fault-tolerant centralized scheduler; however, we note that Clockwork's predictable worker design will make pernicious phenomena like grey failure [19, 30] far easier to detect.

Other benefits of predictability. Concentrating choice makes it easier to implement other guarantees, such as SLOs related to burstiness or per-request cost. The Azure trace in our evaluation, for instance, contained regular, periodic spikes; exploiting advanced knowledge is an appealing future avenue for Clockwork. A further benefit of predictable system components is *performance clarity* [45]: performance bottlenecks and upcoming tasks are easy to reason about in Clockwork. Clockwork's controller also provides a central point for *explanation*, since the controller has complete visibility of the expected and actual request behavior.

8 Related Work

In §6.1 we directly compared Clockwork to Clipper [11] and INFaaS [48]; here we provide additional comments. Both Clipper and INFaaS act are designed as wrappers around existing model execution frameworks: Clipper, in order to provide a unifying abstraction; INFaaS, in order to exploit heterogeneous execution strategies. Being agnostic to the underlying execution engine sacrifices predictability and control over model execution. Both systems treat latency SLOs as long-term, reactive targets; by contrast, Clockwork is explicitly designed to consolidate choice, and exploit predictability by making proactive decisions. Clipper and INFaaS propose several orthogonal concepts that are compatible with Clockwork. Clipper's model selection layer could be superimposed on Clockwork. INFaaS's model variant concept could be integrated into Clockwork; we found similar predictability properties held for DNNs executing on dedicated CPU cores.

Several other projects investigate model serving in virtualized cloud environments and on serverless platforms, where predictability is in the hands of the cloud provider [7, 36, 65]. In industry, TFS^2 [42] is a proprietary model hosting service at Google, about which public information is not available. Amazon SageMaker [49] and Google AI Platform [18] are public cloud DNN serving systems with a similar interface to Clockwork: upload your model, then make inference requests. Both use containers under the cover as an isolation mechanism, and users suffer the associated cold-start latency. Beyond these details, further design information is not publicly known.

Individual DNN inferences are the atomic unit of work for Clockwork. Increasingly, modern ML applications are composed of pipelines or cascades of DNNs [27, 35, 36, 52]. For these applications, performance predictability is strongly desired. We believe there are opportunities to leverage Clockwork's properties to perform more sophisticated pipeline scheduling that provides end-to-end guarantees.

9 Conclusion

As DNN inferences become increasingly central to interactive applications, the requirements for fast response tighten, the volume of requests expands, and the number of models grows. Our model serving system, Clockwork, meets these challenges. Clockwork efficiently fulfills aggressive tail-latency SLOs while supporting thousands of DNN models with different workload characteristics concurrently on each GPU, and scaling out to additional worker machines for increased capacity. The system also successfully isolates models from performance interference caused by other models served on the same system. Our results derive from our design methodology of recursively ensuring all internal architecture components have predictable performance by concentrating all choices in the centralized controller. Notably, our approach required us to either circumvent canonical best-effort mechanisms or orchestrate them to become predictable and illustrates how our design principles can be applied in practice to achieve predictable performance.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Kudlur Manjunath, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. (§5.1).

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. (§2).

[3] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. LASER: A Scalable Response Prediction Platform for Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM)*, 2014. (§2).

[4] Saamer Akhshabi and Constantine Dovrolis. The Evolution of Layered Protocol Stacks leads to an Hourglass-Shaped Architecture. In *Proceedings of the 2011 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011. (§2).

[5] Allied Market Research. Global machine learning chip market to garner $37.85 Billion by 2025, at 40.8% CAGR. `https://www.globenewswire.com/news-release/2020/02/18/1986370/0/en/Global-Machine-Learning-Chip-Market-to-Garner-37-85-Billion-by-2025-at-40-8-CAGR.html`, February 2020. (§2).

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. (§3).

[7] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. In *Proceedings of the 7th IEEE International Conference on Cloud Engineering (IC2E)*, 2019. (§6.5 and 8).

[8] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S Lam. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International World Wide Web Conference (WWW)*, 2017. (§2).

[9] Wai Chee Yau. How Zendesk Serves TensorFlow Models in Production. `https://medium.com/zendesk-engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b`, February 2017. (§2).

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. (§2, 5.1, and 1).

[11] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. (§1, 2, 5.1, 6.1, and 8).

[12] 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 (KDD)*, 2014. (§2).

[13] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013. (§2 and 3).

[14] Christina Delimitrou and Christos Kozyrakis. Amdahl's Law for Tail Latency. *Communications of the ACM*, 61(8):65–72, 2018. (§3).

[15] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–26, 2012. (§3).

[16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. (§7).

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. (§2).

[18] Google AI Platform. Retrieved May 2020 from `https://cloud.google.com/ai-platform/`, 2020. (§1, 2, and 8).

[19] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018. (§7).

[20] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, et al. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020. (§2, 6.5, and 1).

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2018. (§1 and 2).

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. (§1).

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 2016. (§1).

[24] Jeremy Hermann and Mike Del Balso. Meet Michelangelo: Uber's Machine Learning Platform. `https://eng.uber.com/michelangelo/`, September 2017. (§2).

[25] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for MobileNetV3. In *Proceedings of the IEEE 2019 Conference on Computer Vision (ICCV)*, 2019. (§1).

[26] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017. (§1).

[27] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying Large Video Dataset with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. (§8).

[28] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-Excitation Networks. In *Proceedings of the IEEE 2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. (§1).

[29] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *Proceedings of the IEEE 2017 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. (§1).

[30] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017. (§7).

[31] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013. (§3).

[32] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving Deep Learning Models in a Serverless Platform. In *Proceedings of the 6th IEEE International Conference on Cloud Engineering (IC2E)*, 2018. (§6.5).

[33] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. *Chapter 4 - Service Level Objectives*. O'Reilly Media, 2016. `https://landing.google.com/sre/sre-book/chapters/service-level-objectives/`. (§2).

[34] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017. (§2 and 2).

[35] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment*, 10(11), 2017. (§8).

[36] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, 2019. (§8).

[37] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the*

IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR), 2016. (§1).

[38] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014. (§2 and 3).

[39] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020. (§1, 2, 1, and 7).

[40] Neural Network Exchange Format (NNEF). Retrieved May 2020 from `https://www.khronos.org/nnef/`, 2020. (§2).

[41] NVIDIA TensorRT. Retrieved May 2020 from `https://developer.nvidia.com/tensorrt`, 2020. (§5.1).

[42] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on ML Systems at NeurIPS 2017*, 2017. (§1 and 8).

[43] Open Neural Network Exchange Format: The new open ecosystem for interchangeable AI models. Retrieved May 2020 from `https://onnx.ai/`, 2020. (§2).

[44] The ONNX Model Zoo. Retrieved May 2020 from `https://github.com/onnx/models`, 2020. (§1).

[45] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017. (§7).

[46] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. (§2 and 3).

[47] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014. (§7).

[48] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: A Model-less Inference Serving System. *arXiv preprint arXiv:1905.13348*, 2019. (§1, 2, 5.1, 6.1, and 8).

[49] Deploying a Model on Amazon SageMaker Hosting Services. Retrieved May 2020 from `https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-hosting.html`, 2020. (§8).

[50] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *arXiv preprint arXiv:1907.10597*, 2019. (§2).

[51] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC '20)*, 2020. (§6.5).

[52] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019. (§8).

[53] Julien Simon. Amazon Elastic Inference – GPU-Powered Deep Learning Inference Acceleration. `https://aws.amazon.com/blogs/aws/amazon-elastic-inference-gpu-powered-deep-learning-inference-acceleration/`, November 2018. (§2).

[54] Kacper Sokol and Peter A Flach. Glass-box: Explaining ai decisions with counterfactual statements through conversation with a voice-enabled virtual assistant. In *IJCAI*, pages 5868–5870, 2018. (§2).

[55] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*, 2019. (§2).

[56] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE 2015 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. (§1).

[57] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE 2016 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. (§1).

[58] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Gregory Chockler, Haoyuan Li, and Yoav Tock. Dr. Multicast: *Rx* for Data Center Communication Scalability. In *Proceedings of the 5th European Conference on Computer systems (EuroSys)*, 2010. (§3).

[59] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal Segment Networks: Towards Good Pratices for Deep Action Recognition. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, 2016. (§1).

[60] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at Facebook: Understanding inference at the edge. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019. (§1 and 2).

[61] Bin Xiao, Haiping Wu, and Yichen Wei. Simple Baselines for Human Pose Estimation and Tracking. In *Proceedings of the 16th European Conference on Computer Vision (ECCV)*, 2018. (§1).

[62] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE 2017 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. (§1).

[63] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep Layer Aggregation. In *Proceedings of the IEEE 2018 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. (§1).

[64] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. *arXiv preprint arXiv:1605.07146*, 2016. (§1).

[65] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting Cloud Services for Cost-Effective, SLO-aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019. (§8).

[66] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. ResNeSt: Split-Attention Networks. *arXiv preprint arXiv:2004.08955*, 2020. (§2, 2, and 1).

A Model Listing

Model Family	**Model**	**IO Size (kB)**		**Weights**		**GPU Execution Latency (ms)**				
		Input	**Output**	**Size (MB)**	**Transfer (ms)**	**B1**	**B2**	**B4**	**B8**	**B16**
DenseNet [29]	densenet121	602	4	31.8	2.59	3.80	4.52	6.55	10.22	17.91
	densenet161	602	4	114.7	9.33	7.66	10.11	15.13	23.94	40.04
	densenet169	602	4	56.5	4.50	5.18	6.29	8.57	12.82	21.85
	densenet201	602	4	80.0	6.52	6.84	8.45	11.95	18.30	31.03
DLA [63]	dla34	602	4	64.9	5.29	3.06	4.77	7.11	10.66	15.98
GoogLeNet [56]	googlenet	602	4	26.5	2.16	1.54	1.94	2.69	4.19	7.11
Inception v3 [57]	inceptionv3	1073	4	95.3	7.77	4.46	6.85	10.99	16.45	26.17
	xception	602	4	159.3	12.99	4.49	6.64	10.46	18.53	34.55
Mobile Pose [61] + MobileNet [25, 26]	mobile_pose_mobilenet1.0	590	209	20.0	1.63	0.99	1.72	2.99	5.67	10.78
	mobile_pose_mobilenetv3	590	209	19.0	1.55	1.29	1.92	3.13	5.71	11.62
	mobile_pose_resnet18_v1	590	209	51.4	4.19	1.43	2.25	3.52	6.29	11.46
	mobile_pose_resnet50_v1	590	209	102.2	8.31	3.29	5.42	9.00	16.28	29.92
	simple_pose_resnet18_v1b	590	209	61.5	5.00	2.46	3.62	6.67	10.70	18.98
ResNeSt [66]	resnest14	602	4	42.4	3.45	2.70	4.07	6.72	12.61	22.91
	resnest26	602	4	68.2	5.56	4.30	6.07	9.85	18.26	32.52
	resnest50	602	4	109.8	8.93	6.96	9.47	14.27	29.94	56.02
	resnest101	602	4	192.9	15.71	12.31	16.23	25.79	44.65	78.17
ResNet [22]	resnet18_v1	602	4	46.7	3.81	1.27	1.86	2.73	4.06	7.02
	resnet18_v1b	602	4	46.7	3.81	1.25	1.71	2.37	3.93	6.83
	resnet34_v1	602	4	87.2	7.11	2.40	3.39	4.62	7.76	14.40
	resnet34_v1b	602	4	87.2	7.11	2.37	3.37	4.59	7.76	13.32
	resnet50_v1	602	4	102.3	8.33	2.61	3.78	5.61	9.13	15.67
	resnet50_v1b	602	4	102.1	8.33	2.77	3.95	5.88	9.78	16.58
	resnet50_v1c	602	4	102.2	8.31	2.82	4.07	6.11	10.17	17.26
	resnet50_v1d	602	4	102.2	8.31	2.78	4.02	6.01	10.06	17.13
	resnet50_v1s	602	4	102.6	8.35	3.04	4.47	6.99	11.66	20.39
	resnet50_tuned_1.8x	602	4	88.1	7.16	2.24	3.05	4.25	6.65	11.13
	resnet101_v1	602	4	178.3	14.54	5.27	7.62	11.07	18.04	30.30
	resnet101_v1b	602	4	178.0	14.46	5.41	7.80	11.33	18.64	31.18
	resnet101_v1c	602	4	178.1	14.47	5.47	7.91	11.53	19.03	31.98
	resnet101_v1d	602	4	178.1	14.47	5.42	7.87	11.44	18.94	31.84
	resnet101_v1s	602	4	178.5	14.51	5.70	8.35	12.43	20.55	35.10
	resnet101_tuned_1.9x	602	4	136.3	11.08	3.85	5.61	7.47	12.56	20.61
	resnet101_tuned_2.2x	602	4	131.0	10.65	3.72	5.23	7.01	11.28	18.55
	resnet152_v1	602	4	240.9	19.58	7.71	11.14	16.21	26.48	44.60
	resnet152_v1b	602	4	240.5	19.54	7.86	11.36	16.41	27.05	45.49
	resnet152_v1c	602	4	240.5	19.55	7.90	11.48	16.64	27.42	46.24
	resnet152_v1d	602	4	240.5	19.55	7.89	11.45	16.59	27.38	46.01
	resnet152_v1s	602	4	241.0	19.58	8.15	11.91	17.50	28.95	49.27

Table 1: Models used for Clockwork experiments. Pre-trained models were sourced from the ONNX Model Zoo [44] and the GluonCV Model Zoo [20], and optimized for NVIDIA Tesla v100 GPUs using TVM v0.7 [10] *Continues on next page.*

Model Family	Model	IO Size (kB)		Weights		GPU Execution Latency (ms)				
		Input	**Output**	**Size (MB)**	**Transfer (ms)**	**B1**	**B2**	**B4**	**B8**	**B16**
ResNet v2 [23]	resnet18_v2	602	4	46.7	3.81	1.32	1.81	2.48	4.42	7.12
	resnet34_v2	602	4	87.2	7.11	2.55	3.44	4.83	7.90	14.01
	resnet50_v2	602	4	102.2	8.32	2.73	4.05	5.87	9.93	17.3
	resnet101_v2	602	4	178.1	14.47	5.51	8.05	11.83	18.14	33.57
	resnet152_v2	602	4	240.6	19.56	8.21	11.66	17.03	27.60	48.54
ResNeXt [62]	resnext50_32x4d	602	4	100.0	8.15	2.18	3.23	5.35	9.21	17.42
	resnext101_32x4d	602	4	176.4	14.34	4.65	6.27	10.06	17.75	32.83
	resnext101_64x4d	602	4	333.4	27.18	6.46	10.24	17.13	30.42	60.23
SENet [28]	se_resnext50_32x4d	602	4	110.1	8.95	3.20	4.47	6.87	11.50	20.64
	se_resnext101_32x4d	602	4	195.5	15.89	6.23	8.24	12.53	21.02	37.89
	se_resnext101_64x4d	602	4	352.5	28.75	8.18	12.97	19.93	34.99	66.44
TSN [59]	tsn_inceptionv1_kinetics400	1073	1.6	24.0	1.96	1.95	2.76	4.44	7.51	13.43
	tsn_inceptionv3_kinetics400	1073	1.6	90.4	7.37	4.47	6.87	10.97	16.43	26.12
	tsn_resnet18_v1b_kinetics400	602	1.6	45.5	3.71	1.25	1.72	2.38	3.93	6.83
	tsn_resnet34_v1b_kinetics400	602	1.6	85.9	7.01	2.38	3.38	4.59	7.74	13.37
	tsn_resnet50_v1b_kinetics400	602	1.6	97.2	7.93	2.77	3.94	5.85	9.77	16.52
	tsn_resnet101_v1b_kinetics400	602	1.6	173.1	14.11	5.42	7.80	11.30	18.63	31.15
	tsn_resnet152_v1b_kinetics400	602	1.6	235.6	19.21	7.87	11.35	16.42	27.07	45.44
Wide ResNet [64]	cifar_wideresnet16_10	12	0.04	68.5	5.59	1.27	1.72	2.61	4.07	7.62
	cifar_wideresnet28_10	12	0.04	145.9	11.93	2.21	3.57	5.42	8.41	16.05
	cifar_wideresnet40_8	12	0.04	143.0	11.69	2.49	3.90	5.99	9.86	17.14
Winograd [37] + ResNet v2 [23]	winograd_resnet18_v2	602	4	77.4	6.31	0.95	1.17	1.71	2.81	5.09
	winograd_resnet50_v2	602	4	128.7	10.49	3.39	4.24	6.07	10.28	18.84
	winograd_resnet101_v2	602	4	235.8	19.23	6.36	7.71	10.71	17.26	33.52
	winograd_resnet152_v2	602	4	324.1	26.42	9.40	11.13	15.92	24.42	28.92

Table 1: *Continues from previous page.* Models used for Clockwork experiments. Pre-trained models were sourced from the ONNX Model Zoo [44] and the GluonCV Model Zoo [20], and optimized for NVIDIA Tesla v100 GPUs using TVM v0.7 [10].

B Detailed Scheduler Implementation

The basic Clockwork scheduler creates a schedule out of a single, global request queue. To improve the quality of the schedule, it postpones the creation of actions as long it stays within SLO limits. The scheduler also ensures a minimal amount of work is carried out by each worker at any moment.

Strategies. Each request generates several *strategies* for its execution, one for each batch size supported by the model. The strategy specifies a *required start time*, calculated using the request's deadline and the estimated execution time for the model at that batch size. Strategies are enqueued into a single strategy queue shared by all models and workers, ordered by required start time. Larger batch sizes have an earlier required start time, and are therefore dequeued first.

Selecting Strategies. Each INFER executor is given enough actions to keep the executor busy for the next 5ms. The scheduler iterates the strategy queue, skipping strategies for models that not loaded on the executor; for batch sizes that are too big (*i.e.*, for which the model has insufficient requests); for batch sizes that are too small (*i.e.*, for which there are enough requests for a larger batch size); and for strategies that cannot complete in time given the executor's current outstanding work. When an eligible strategy is found, requests from that model are batched into a single INFER action. This invalidates all remaining strategies for the selected requests; those strategies are dropped from the queue. Strategies are also dropped if their required start time has passed. This approach ensures that large batch sizes are selected where possible, and if given the choice between a large batch size or a small one, the larger batch size is chosen - possibly at the expense of dropping an unfortunately timed earlier request.

Loading Models. Each LOAD executor is also given enough actions to keep it busy for the next 5ms. The scheduler must decide whether to load a model, and if so, which. To do this, the scheduler maintains and incrementally updates *load* and *demand* estimates across models and GPU. :

- d_m the total demand for each model m
- $a_{m,g}$ the demand allocation of model m on GPU g.
- $\ell_g = \sum_m a_{m,g}$ the total load on each GPU g

A model's total demand d_m is the total estimated execution time of the model's outstanding requests; it is updated when requests for that model arrive and complete. The demand allocations for a model are also updated when requests arrive and complete; they are calculated such that $\sum_g a_{m,g} = d_m$. Demand allocations are 0 for GPUs where the model is not loaded. For GPUs where the model is loaded, demand allocations are inversely proportional to the GPU's load, since overloaded GPUs will be able to execute proportionally less of the total demand. Lastly, each GPU's total load ℓ_g is the sum of its allocations across all models.

Using the above estimates, each model's load priority is calculated as $p_{m,g} = d_m - \sum_g a_{m,g} \times \frac{\text{capacity}_g}{\ell_g}$. A model's load priority is an estimate of its unfulfilled work. For models that are not loaded on any GPUs, its priority is equal to its outstanding work. For a model loaded on a GPU that is mostly idle, its load priority is negative, since the GPU can serve more work than the model demands.

Clockwork does not attempt to converge to a perfect demand allocation each time the system's state changes. Instead Clockwork incrementally updates each model's demand allocation and load priority, when:

- new requests arrive for that model;
- an INFER is initiated for that model;
- when LOAD and UNLOAD affect a model; and
- when a request passes the point it can benefit from LOAD before its deadline.

The scheduler selects LOAD actions by choosing the highest priority model that is not already loaded. Clockwork does not load models with negative priority, which indicates the model's demands are fulfilled. To select models for UNLOAD, Clockwork uses a least-recently-used (LRU) policy.