Navigator: A Decentralized Scheduler for Latency-Sensitive AI Workflows

Yuting Yang
dept. Computer Science
Cornell University
Ithaca, USA
yy354@cornell.edu

Andrea Merlina
dept. Informatics
University of Oslo
Norway
andremer@ifi.uio.no

Weijia Song
dept. Computer Science
Cornell University
USA
ws393@cornell.edu

Tiancheng Yuan
dept. Computer Science
Cornell University
USA
ty373@cornell.edu

Ken Birman
dept. Computer Science
Cornell University
USA
ken@cs.cornell.edu

Roman Vitenberg
dept. Informatics
University of Oslo
Norway
romanvi@ifi.uio.no

2024 IEEE International Conference on Edge Computing and Communications (EDGE) | 979-8-3503-6849-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/EDGE62653.2024.00015

***Abstract*—We consider AI inference and generative query processing in distributed systems: an increasingly popular edge-computing use case. In such systems, coscheduling of GPU memory management and task placement represents a promising opportunity. We propose Navigator, a novel framework that unifies these functions to reduce job latency while using resources efficiently, placing tasks where data dependencies will be satisfied, collocating tasks from the same job (when this will not overload the host or its GPU), and efficiently managing GPU memory. Comparison with other state of the art schedulers shows a significant reduction in completion times while requiring the same amount or even fewer resources. In one case, just half the servers were needed for processing the same workload.**

I. Introduction

We consider applications in which a rack of compute servers is deployed physically close to data sources (cameras, microphones, etc). There is growing interest in this model because large data objects can be costly to upload to a cloud, and because it enables the first steps of an AI application to discard uninteresting data without first uploading everything (and often, storing all of that data within the cloud). To amortize the costs of the rack, resources such as GPUs can be shared across multiple applications. Edge computing need not be the whole story: some sensors can perform non-trivial computation [91], [92], [81], and the edge will often be coupled to a cloud where follow-on steps could run [3], [4], [58], [64], [93]. Nonetheless, edge servers are in a position to play roles that neither smart sensors nor cloud applications could perform.

Interactive AIs are programs (or distributed pipelines of programs) triggered by some sort of initial event and that finish by reporting a result or taking an action, unlike the forward-and-forget pattern seen in asynchronous streaming models. We treat each event as a new job, modelling it as a control-flow graph annotated by data dependencies, size estimates and home locations (storage servers). The nodes in the graph represent AI programs, and a distribution of expected runtimes is assumed to be available for each. Given this context, Navigator optimizes to minimize interactive delay while using host and GPU resources efficiently.

Our approach gives Navigator control over GPU caching and main-memory caching, tracking server loads and cache use and deciding what to retain in cache, what to eject, and where the computations needed for each step of each job should run. This enables data-dependency aware scheduling, which is important because AI models range from hundreds of megabytes to many gigabytes in size [75]. At these sizes, a single cache miss can have huge latency consequences.

To reduce delays on the performance-critical path Navigator employs a decentralized scheduling model, in which any worker node (compute server) can schedule tasks on any other worker. This removes a query to a centralized scheduler from the critical path but departs from the way that most existing schedulers behave [38], [55], [52], [49], [68], [50]. Centralization is generally assumed to enable better decision-making, but our evaluation will show that even though a decentralized algorithm requires replicating the data used in the scheduler, the overheads of doing so can be kept low by limiting the frequency of updates and leveraging a modern data replication package [39], and that the resulting scheduling quality is as good or better than seen with centralized scheduling. We see this as one surprising contribution of our work. Other contributions include:

1) A new decentralized QoS-oriented scheduler for AIs coded in PyTorch [36], TensorFlow [21], or MXNet [47].
2) Dependency-aware GPU cache and memory management.
3) Distributed tracking of server loads and GPU cache contents, enabling decentralized scheduling.
4) Experiments and simulation showing that Navigator can reduce latency by 2x-6x relative to other approaches.

II. Deployment Scenarios and Environment

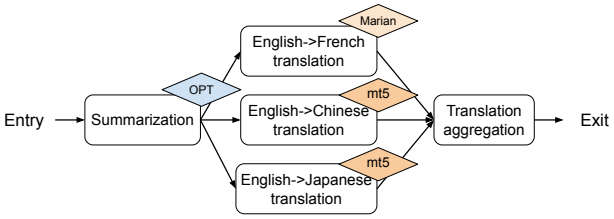
A. Dataflow Graphs

Navigator's data flow graph (*DFG*) representation is similar to that used in prior work [12], [18], [78], [55]. As noted earlier, these are directed acyclic graphs $G = (E, V)$, where vertex $v \in V$ represents an AI computation. Edge $e \in E$ represents precedence constraints: output from the upstream AI will become input to the downstream AI. We attach a diamond

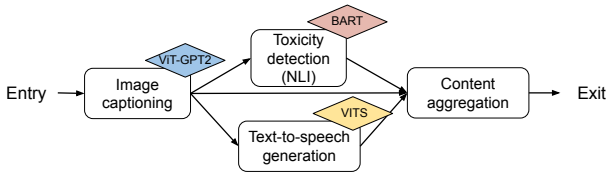
979-8-3503-6849-9/24/$31.00 ©2024 IEEE
DOI 10.1109/EDGE62653.2024.00015

box to vertex v to represent data dependencies on the AI model and other objects that v requires to perform its computational task. As seen in Figure 1 different colors are used for different objects, while identical colors denote dependency upon the identical objects. Our figures omit object sizes and estimated job/task execution times but the *DFG* includes them.

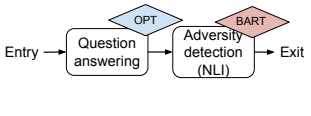
We will use four examples of *DFG*s throughout this paper. The workflow in Figure 1a autocaptions for multilingual online meetings [85], aggregating the translations as a single output. It uses Meta's opt-1.3b Open Pre-trained Transformer (OPT) model to process input [77], Marian [33], [34] for French translation, and mt5 [62], [41] for both Chinese and Japanese (mt5 plays two roles but uses a single model). To annotate this and our other *DFG*s with expected runtimes and model sizes, we profiled representative test cases, then adopted parameters covering 95% or more of the observed data.



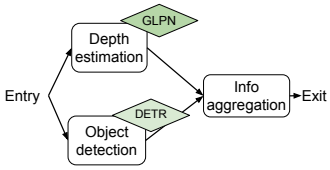
(a) language translation pipeline



(b) image reading pipeline



(c) Q&A dialogue pipeline



(d) 3D perception pipeline

Fig. 1: For AI Pipelines

The second workflow (Figure 1b) arises in an application that auto-captions images for children's education [19]. The *DFG* employs the ViT-GPT2 model [87], [88] for automated captioning, ESPnet [30] for vocalization, and BART [44] to ensure that only child-safe results are generated.

The third workflow (Figure 1c) might be used by a Virtual Personal Assistant (VPA) [27]; it uses the same OPT model but with prompts to "shape" the desired output, configuring BART to target an adult rather than a child.

The fourth workflow (Figure 1d) arises in an assistive application for a vision-impaired user. Object detection is performed by the DEtection TRansformer (DETR) [53] model and depth estimation using a hierarchical transformer encoder-decoder model [82]. The final vertex combines these two estimates.

B. Deployment Assumptions

Navigator can be used both in cloud datacenters and on edge clusters located near data-capture devices [73]. GPU support is assumed, but edge devices are limited to inexpensive GPUs: a single GPU memory will often be too small to hold the full set of active AI models. For example, if all the AIs in Figure 1 were in simultaneous use, the aggregated memory size for AI models would be nearly 35GB: substantially more than a single GPU could hold, underscoring that GPU memory management will be a priority.

C. Scheduler Objectives

Our four example AI workflows all require fast responses, and this is typical of edge AI. For example, when performing language translation and image generation, end-to-end performance of the *DFG* determines the wait time the human user will perceive. Navigator is expected to minimize end-to-end latency, defined as the difference between the end time (when the processing of the exit task is finished) and the start time (when the job instance is generated). Optimal minimization of latency in *DFG* structured workflows is NP-complete, hence Navigator depends on heuristics [11], [13], [9]. We do not optimize for resource consumption or energy, but Section VI-D will show that our algorithm is parsimonious in both respects.

Our scheduling task differs from that addressed by existing schedulers, which often focus on ML training scenarios [57], [73], [59], [56]. ML training is highly iterative with long-running distributed jobs. Data objects will be accessed again and again from cache, amortizing any initial fetch delays. In our setting, GPU memory isn't large enough to hold every active AI model, and delays to fetch a missing model count against QoS objectives. Thus, new techniques are needed.

III. System Architecture

As shown in Figure 2, we assume that each worker consists of a host supporting storage, host computation, and a GPU unit for heavy lifting. All workers run Navigator, which is a fully decentralized and symmetric: every worker has visibility of the system state and can assign tasks to any of its peers.

Job instances are scheduled and executed by workers at the granularity of *tasks*. A client triggers a new job by sending a request containing an input object to one of the workers, which

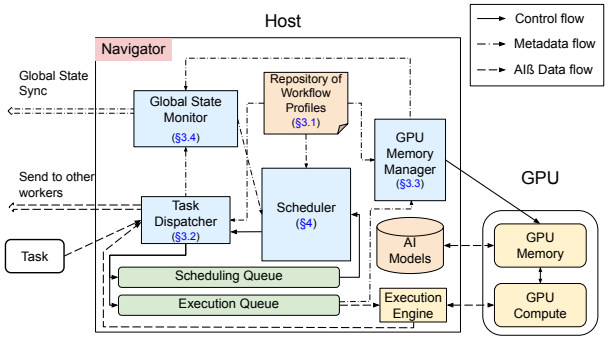


Fig. 2: Worker Components in Navigator

launches the ingress task. As its first step, this task creates an *Activated Dataflow Graph* (*ADFG*) for the job instance. The *ADFG* is based on the *DFG*; it is a map containing initial worker assignment for all tasks in the job instance. However, the assignment can still be dynamically adjusted, as we explain below. Once created, the *ADFG* is piggybacked from task to task as the job executes.

The architecture of Navigator is seen in Figure 2 and encompasses five components. (1) Workflow Profiling collects profiled information about the vertices in a *DFG* such as average execution times. (2) The Decentralized Global State Monitor collects and periodically exchanges information about all the workers: their load, cache bitmap, etc. Navigator's Scheduler component creates a *ADFG* and later adjusts it. (3) The task dispatcher executes the decisions taken by the Scheduler, sending task start requests to the correct worker as the prior task(s) complete. (4) The GPU Memory Manager fetches models on the fly and manages a GPU cache. (5) Task processing is managed by the Execution Engine, which includes a plug-in customized for each supported AI framework. We now describe each of the components, while the details of Navigator's scheduling algorithm are presented in Section IV.

A. Repository of Workflow Profiles

Navigator's Workflow Profiles Repository holds meta-information about *DFG*s. In addition to static information (expected runtime costs and input/output object sizes), each task is annotated with the actual sizes of known inputs.

B. Scheduler and Task Dispatcher

The flow of job handling is illustrated in Figure 3. When a worker w receives a job processing request from a client, the request is appended to *scheduling queue* on w. The Scheduler component loops, processing the first request on the *scheduling queue*. For each request, it produces an *ADFG* which includes the worker assignment for each task.

Once the *ADFG* has been created, Task Dispatcher on W sends it to worker v on which the entry task in *ADFG* is scheduled for execution (v may happen to be w). Upon receiving the *ADFG*, v appends the entry task to its *execution queue*. The Task Dispatcher on v also loops, examining the first task t on the *execution queue*, popping it from the queue

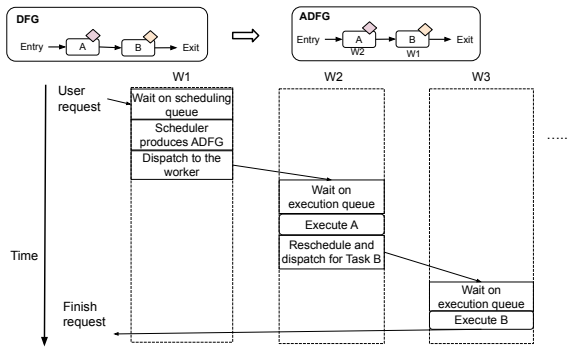


Fig. 3: Example of Job Instance Handling

if it can be executed, and then repeating as soon as the worker has adequate resources to process a new task.

The determination of readiness for execution entails verifying that all inputs to t produced by predecessor tasks in the *ADFG* are available. If not, Task Dispatcher temporarily leaves t in the queue and proceeds to the next task.

Once the prerequisites for executing t are satisfied, the next step entails ensuring that the AI model is in GPU memory. If the model is not in the GPU cache, it will be transferred in from host memory. While the transfer is underway, Task Dispatcher leaves t in the queue and proceeds to the next task.

Once the model is in the GPU cache, the Execution Engine starts processing t by doing an upcall to the task logic in the relevant framework. When the execution is finished, if necessary, the Scheduler adjusts the worker assignment for successors of t in the *ADFG* using the algorithm presented in Section IV. After the adjustment, Task Dispatcher sends the *ADFG* along with the output data of t to workers assigned to execute the successors of t.

Preliminary scheduling decisions enable Navigator to collocate a series of tasks on the same worker. This saves time by avoiding the need to move the output of one task to the location where the next worker would run. Moreover some aspects of task assignment can only occur at the outset: if a set of tasks are followed by a join, they would have no way to make a coordinated assignment for the join task. But the unpredictability of actual runtimes and object sizes can create situations in which rigidly adhering to initial scheduling decisions could overload a worker. Accordingly, we adopted a two-step scheduling approach in which the initial *ADFG* can be adjusted during runtime. This turns out to be essential (see Section VI-C1).

C. GPU Memory Manager

The purpose of this component is to manage AI models on worker nodes: the *Navigator cache* and the GPU model execution memory. *Navigator cache* holds reusable model objects in compressed form, whereas the execution memory is used to hold input objects for tasks and a decompressed model for each currently-active task.

As mentioned in Sections I and II-B, GPU memories will often be too small to simply keep a copy of every AI model that might be needed. On the other hand, it is costly to fetch large models at the last instant when a task will be executed. Fortunately, AI models are reusable across different instances of the same pipeline and even across different pipelines, as we saw in Figure 1 where a single model was able to produce output in multiple languages. Retention of models in GPU memory can thus have a significant impact on job completion times. Prior work has often employed a static model placement, designating some set of workers that will "serve" each model, and managing GPU memory using a policy that the AI logic itself ultimately controls. In contrast, we take the approach of scheduler-triggered GPU memory management, in which the scheduler decides which worker will run each task, at which point the worker itself makes local decisions about

model placement (both fetching and eviction) based on its assigned tasks. This design is especially beneficial for dynamic workloads with high variation or a sudden burst of requests for the same pipeline, because the Navigator scheduler can dynamically add additional workers to accommodate bursts of demand for particular AIs. In the experiments in Section VI, we can see that proper task assignment depends upon careful management of model placement, and that Navigator is able to adjust to workloads with variations.

D. Global State Monitor

Navigator's Global State Monitor manages a distributed table of system state. It runs on all workers and holds per-worker queue processing time and GPU memory (cache) contents. The queue processing time is the time to complete all tasks currently on the worker's *execution queue*. The Navigator cache information includes a bitmap specifying which models are in cache and also tracks the remaining unused cache capacity.

To distribute updates, we take advantage of a modern multicast protocol optimized to leverage fast networking [39]. Each worker periodically multicasts updates about its local worker's state information to all other workers. However, we limit the frequency of these updates to ensure that overhead will be low. The staleness of the information a worker sees about other workers has an upper bound that is equal to the dissemination interval. In the experiments at Section VI, we showed the scheduling mechanism has a high tolerance of information staleness, and enabled us to configure the update frequency to ensure that state information remains sufficiently accurate for Navigator's task assignment decisions.

IV. Navigator Scheduler Design

Navigator employs a heuristic scheduling mechanism with dynamic coordination between workers. The mechanism exhibits an important difference from schedulers in prior literature in that Navigator manages GPU memory and balances task assignment using a metric that reflects AI model object placements. The experiments reported in Section VI confirm that this balance significantly improves end-to-end latency for job instances.

As explained in Section III-B, scheduling consists of two phases. The job instance planning phase produces the initial *ADFG*, and is described in Section IV-B. The dynamic adjustment phase is presented in Section IV-C.

A. Parameters

The Navigator algorithm uses static information from the job *DFG*, the *ADFG* data produced by the initial scheduler, information from the repository of workflow profiles, and real-time information about worker states provided by Global State Monitor.

Task parameters The expected runtime of task t on worker w, $R(t,w)$, as well as the expected task input size $|input_t|$ and output size $|output_t|$ are obtained from the repository of workflow profiles. The transfer duration $TD_{input}(t)$ for a task input is estimated via object size and network transmission capacity. The duration of transfer between two different workers is estimated by a commonly accepted heuristic as $TD_{input}(t) = |input_t|/network\ transmision\ capacity + \delta_{network}$, where $\delta_{network}$ is a constant factor contributing to the latency [65]. This estimate is used when determining whether it would be more advantageous to schedule a successor to task t on the same node, or to schedule it on a different node where the needed AI models will be local. Tasks may also be executed on different nodes if scheduling a successor on the same node would force the latter task to wait because of resource oversubscription.

AI model parameters For AI model m, the model size $|m|$ is also stored in the repository of profiles. This information is used to estimate the time to fetch the model from host machine to GPU memory as follows: $TD_{model}(m,w) = |m|/PCIe\ transmission\ capacity_w + \delta_{PCIe}(w)$. It is also used to estimate the available memory that will remain in *Navigator cache* on the GPU after fetching the model.

Worker parameters The Global State Monitor collects and disseminates information about worker load and backlogs. Each worker w estimates the time ($FT(w)$) that would be required to fully execute all the tasks on its current *execution queue*. This is computed as $FT(w) = current\ time + \sum_{t \in execution\ queue} R(t,w)$. We similarly define $FT(t,w)$ as the estimated finish time of task $t \in$ *execution queue* on worker w. The available memory in the *Navigator cache* of worker w is denoted $AVC(w)$. It is derived from the cache size and the cumulative size of the models currently in the cache: $AVC(w) = gpu\ capacity_w - \sum_{m \in \text{the cache of } w} |m|$.

B. Planning Phase

The task assignment algorithm used during Navigator's job planning phase maps tasks in a job instance to workers. This algorithm is inspired by a well-researched process scheduling algorithm, Heterogeneous Earliest Finish Time (HEFT) [10]. Like HEFT, Navigator uses an upward ranking to prioritize the task scheduling order and selects the workers with the earliest start time. However, Navigator extends the classic HEFT in several respects. HEFT does not consider the load of a machine when performing the scheduling, which in our setting could result in inefficient task assignment. Additionally, HEFT lacks mechanisms to factor in the potential benefits of co-location of the task with cached model objects on which it depends, or with input data. A third issue arises because HEFT locks down a plan for the entire job at the outset, preventing adjustments in the event that worker state changes in a significant way as the job instance proceeds. We address the first two limitations in the Navigator's job planning algorithm and the third in its dynamic adjustment phase.

1) Vertex Ranking

Like HEFT, Navigator starts by assigning a rank to each task t. We use $t \prec t'$ to denote that task t is a direct predecessor of t' in the *DFG*. Since the target worker w is unknown at the time of ranking, $R(t)$ is the average of $R(t,w)$ over the worker set. The rank is defined recursively as

$$rank(t) = R(t) + max_{t \prec t'}(TD_{\text{output}}(t) + rank(t')) \quad (1)$$

Using this ranking, tasks in the same job instance will be prioritized based on their dependencies and impact on the end-to-end latency. A task has a higher priority than a successor that depends on its input. Priorities of tasks with the same level of dependencies will be determined by their execution time and the size of any intermediate data objects that the task requires but are not locally available, because data transmission cost affects the end-to-end latency. In our setting, we see heavy reuse of *DFG*s, resulting in job instances containing identically ranked tasks. In these cases, time of arrival determines the ranking.

Notice that many aspects of task ranking can be performed statically. Navigator carries out these computations just once, when the *DFG* is initially loaded, then saves the results into its repository. Later, when dynamic inputs become available, the statically-computed rankings will merely need to be updated, not recomputed from scratch.

2) Task Assignments

The role of Navigator's job instance planning phase is to produce an *ADFG*, which is a map from task ID to worker ID. The algorithm starts by considering the vertex rankings associated with the *DFG*. Navigator's Scheduler module iterates through the tasks in descending order of priority, carrying out per-task scheduling decisions.

For a task t, this is done by looping through all worker nodes and selecting the worker with the earliest start time. For a given worker w, there are three main constituents, the transfer duration $TD_{model}(t, w)$, the finish time for all queued tasks $FT(w)$, and the finish time for transferring all inputs of t, $FT_{allInputs}(t, w)$.

$TD_{model}(t, w)$ is the time for worker w to load AI model m for use by task t. Leveraging the data managed by the Global State Monitor, the scheduler can determine which GPUs currently hold decompressed instances of each of the models in the *Navigator cache* for all workers in the system. If a task is assigned to a worker that will need to fetch an AI model, the required time is computed as

$$TD_{model}(t, w) = \begin{cases} 0, \text{ if } m_t \in \textit{Navigator cache} \text{ on } w \\ TD_{model}(m, w), \text{ if } m_t \notin \text{cache} \wedge |m_t| \leq AVC(w) \\ TD_{model}(m, w) + \textit{eviction penalty}, \text{ otherwise} \end{cases} \quad (2)$$

Eviction penalty. Consider some task t that needs to be scheduled. Navigator could schedule t on a worker w that already holds needed model m, or could also expand the pool of workers by assigning to some other worker w' where m is not currently in cache. Naively, we should prioritize w over w' if the expected delay at w is longer than the transfer delay for loading m at w' - but this overlooks the likelihood that some other model m' will be evicted from cache by w' to make room for m. The penalty is intended to capture this effect.

$FT(w)$ is the estimated time for worker w to finish all tasks on its *execution queue*. At each run of job instance planning, the scheduler creates a map of finish times for all workers, *worker_FT_map* by fetching the information from Global State Monitor, as shown on line 2 of Algorithm 1. Scheduler also updates the map as it assigns tasks to workers (see Algorithm 1, line 12), since the assignments would affect later tasks in this job instance if scheduled onto the same worker. The planning algorithm incorporates $FT(w)$ to account for the wait time on the workers' queue.

$AT_{allInput}(t, w)$ is the arrival time for all inputs of task t to be received by worker w. It is calculated as the maximum among the arrival times on worker w, for the outputs produced by the predecessors of t. Because the scheduler processes the tasks according to their rankings, when task t is examined its prerequisite tasks will have already been assigned. During job instance planning, Scheduler keeps track of its task assignments, and the estimated finish time of task t on a scheduled worker w. Thus, the time when the output of task t' arrives from worker $ADFG[t']$ at worker w is computed as

$$AT_{input}(t', t, w) = \begin{cases} FT(t', ADFG[t']), \textbf{if } w = ADFG[t'] \\ FT(t', ADFG[t']) + TD_{output}(t'), \text{otherwise} \end{cases} \quad (3)$$

$$AT_{allInputs}(t, w) = max_{t' \prec t} AT_{input}(t', t, w) \quad (4)$$

Algorithm 1 shows the computation steps of the planning algorithm. The complexity of the algorithm is $O(E * W)$, where E is the total number of edges in the *DFG* and W is the total number of worker nodes.

Algorithm 1 Job Planning Algorithm

Input: *DFG*
Output: *ADFG*
1: compute the ranks for all tasks in the *DFG* using Equation 1
2: populate *worker_FT_map* from Global State Monitor // *worker_FT_map* is a map containing $FT(w)$ $\forall w \in$ *workers*
3: *TaskSet* $\leftarrow$ *all tasks* $\in$ *DFG*
4: **while** *TaskSet* $\neq \emptyset$ **do**
5: $t \leftarrow$ a task in *TaskSet* with the biggest rank
6: *TaskSet* $\leftarrow$ *TaskSet* $\setminus t$
7: **for** $w \in$ *workers* **do**
8: $x \leftarrow max($*worker_FT_map[w]*$, AT_{allInputs}(t, w))$
9: $FT(t, w) \leftarrow x + TD_{model}(m_t) + R(t, w)$
10: $w_{min} \leftarrow \arg\min_{w \in workers} FT(t, w)$
11: $ADFG[t] \leftarrow w_{min}$
12: *worker_FT_map*$[w_{min}] \leftarrow FT(t, w_{min})$
13: return *ADFG*

C. Dynamic Adjustment Phase

Whereas job planning occurs when a new job instance is created, dynamic adjustment enables the system to adapt if predicted runtimes, object sizes, or transfer times were estimated inaccurately. Each time the Execution Engine finishes executing some task t, Scheduler will run Algorithm 2. It first checks that t is not a join task: such a task cannot be moved to a different worker without coordination across the predecessor tasks. For a non-join, it reschedules if the wait time on the planned worker exceeds a preconfigured threshold.

If rescheduling is needed, the new worker assignment is performed by ranking workers and selecting the one that would

start the task first. The start time is calculated based on the wait time on the worker's *execution queue*, required model fetching time, and the input transfer duration.

Algorithm 2 Task Dynamic Adjustment Algorithm

Input: $ADFG, t$
Output: *opt_worker*
$w_{planned} \leftarrow ADFG[t]$
above_threshold $\leftarrow FT(w) > R(t,w) * threshold$
require_reschedule $\leftarrow$ (t is not a join task) $\wedge$ *above_threshold*
if $\neg$*require_reschedule* **then**
    return $w_{planned}$
populate *worker_FT_map* from Global State Monitor
for $w \in workers$ **do**
    $x \leftarrow worker\_FT\_map[w]$
    $FT(t,w) \leftarrow x + TD_{model}(t,w) + R(t,w)$
    if w is not this scheduler's worker **then**
        $FT(t,w) \leftarrow FT(t,w) + TD_{input}(t)$
opt_worker $\leftarrow \arg\min_{w \in workers} FT(t,w)$
return *opt_worker*

V. Implementation

Navigator runs as a component of an open-source platform called Cascade [72], [39], which was created to reduce overheads for AI jobs. Prior to our work, Cascade held data (like a key-value store, file system, or message queuing middleware system). It also offered a hosting environment for low-latency AI tasks. Navigator enriches this with AI scheduling.

Cascade objects are simply variable-length byte vectors named by file-system pathnames. Each has a small set of home servers selected using a randomized hash-based object placement within"shards" of size 2 or 3. Access is free on a home server, but a network transfer would occur for an access initiated from some other server. AI models and other inputs to AI tasks would all be Cascade objects.

Cascade additionally includes a storage/retrieval layer for in-memory replication of small data objects. This layer is called the shared state table (SST), and is used by Navigator to disseminate scheduler metadata (see Section V-B).

Lacking Navigator, Cascade offered an automated hash-based load-balancer that was used in situations where a pool of servers all could handle a given request. Integration of Navigator with Cascade centered on replacing this mechanism with the decentralized Navigator scheduler and extending Cascade's preexisting support for GPU accelerators with Navigator's GPU memory management functionality.

A. Data communication layer

1) Remote Direct Access Communication and the Data Plane Developers Kit

Cascade communication is optimized for modern networking, with a focus on two technologies receiving substantial industry attention: RDMA and DPDK. Cascade supports a mix of communication modalities: point-to-point messaging, atomic multicast, and durable updates using a Paxos-based replication model. For example, object replication within a shard uses atomic multicast or Paxos (the former for in-memory performance; the latter for persistence). The SST data structure maps to point-to-point messaging. Each of these, in turn, would be carried out using RDMA or DPDK, depending on which option Cascade is configured to select.

Remote Direct Memory Access (RDMA) [5] is a technology originally created for high-speed computing on bare-metal HPC clusters equipped with Infiniband networks. During the past decade, RDMA has migrated to conventional data centers [61]. The technology is fast, reliable, and offers TCP-like guarantees. By moving the protocol stack into hardware, RDMA frees the host computer to focus on other tasks. However, fully leveraging RDMA requires a substantial rethinking of end-to-end protocols and systems: its speed is closer to that of a computer backplane than a network.

The Data Plane Developers Kit (DPDK) is a pure software solution, and runs over TCP. DPDK moves the TCP stack to user space, bypassing the operating system kernel and by so doing, gaining a significant speedup. In Cascade experiments, DPDK is about twice as fast as TCP and has much lower latencies, but RDMA offers a further factor of two in throughput and a further reduction in latency, particularly for data transfers larger than about 10KB.

2) Data Transfer between Task Dispatchers

Although Navigator's algorithms anticipate object transfers and include cost estimates, Navigator itself does not currently initiate such actions. Instead, the AIs themselves access objects by performing application-layer actions such as opening a file that is in fact hosted by Cascade, reading a key-value object hosted by Cascade, or receiving a pub-sub object that actually is a Cascade object. In all of these cases, Cascade itself will then transparently fetch the object either from a local host cache, from local storage if the server making the access is a home node for the object, or from one of the home nodes if the object is hosted on other servers. Navigator does not peek into the Cascade host cache, but instead adopts the view that every object accessed during an AI job will be in memory somewhere in the system, and also that if the object in question is a large AI model - the case that can become costly - that DMA transfer from the local cache into GPU memory is comparable in speed to RDMA or DPDK transfer from a remote server's memory to the local GPU memory.

Navigator treats one kind of object differently: results of a task that become inputs to a successor task. Here, the distinction is that whereas the outputs of some programs need to be saved and will be reused later, when we consider a *DFG* there may also be transient outputs produced by one stage and then consumed and discarded by its successor stages. Here, the cost depends on where the producer stage runs, and where the consumers run. If an object is small, or if it is large but resides in the GPU cache on a node where the consumer will run, this cost is 0. This form of co-location is clearly very beneficial. Otherwise, Navigator simply assumes that a DMA or RDMA transfer will occur, and models the cost accordingly. The resulting model is presented in Figure 4.

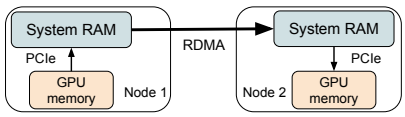


Fig. 4: Network Transfer between Nodes

Navigator Shared State Table(SST) on worker 0

Worker id	Wait_time (ms)	Models in GPU	Available GPU memory(MB)
0	**561**	**[1]**	**12000**
1	1000	[3]	8000
2	0	[3]	8000
3	3000	[4, 5]	1000
4	561	[1]	12000
5	1174	[3]	8000

RDMA

Navigator Shared State Table(SST) on worker 1

Worker id	Wait_time (ms)	Models in GPU	Available GPU memory(MB)
0	0	[ ]	15000
1	**2000**	**[4]**	**9000**
2	0	[3]	8000
3	3000	[4, 5]	1000
4	561	[1]	12000
5	1174	[3]	8000

Fig. 5: Navigator Shared State Table (SST)

B. Shared State Table Considerations

Global State Monitor functionality is realized through Shared State Table mechanism in Cascade. While most objects stored in Cascade are simply key-value tuples with home nodes, but accessible from anywhere, the SST is a very different kind of storage structure. As summarized above, the SST is an extensible table replicated over all servers. Navigator's "instance" of the SST has one row per server, and a fixed row format. The resulting replicated object is an inherently $O(n^2)$ data structure: every node has a copy of the data structure, and when a node updates its row and *pushes* that update, one RDMA write per peer will be issued, to update the corresponding row in that peer. Fortunately, n in our target settings would generally be fairly small: most edge compute clusters have just a few (2 to perhaps 64) blade-style computers in a rack. Even so, we limit the update rate to ensure that overhead is not excessive.

Navigator's decentralized algorithm requires that a node making a scheduling choice be able to estimate the states of other nodes, but does not require perfect accuracy or lock-step coordination. Accordingly, we decided to limit the frequency of SST updates using a parameter in the Navigator configuration file, and then to experimentally determine the lowest acceptable update rate, thereby ensuring that SST overheads will be minimized. In Section VI-B we report on these experiments, confirming that Navigator performs well even with somewhat stale metadata, and justifying an update frequency of 5 pushes per second: negligible given that RDMA can send 75M small messages per second on each network link.

C. GPU Memory Management

Navigator's GPU Memory Manager controls caching for recently used AI models in GPU memory. It performs model fetching and eviction as new tasks are assigned to the associated worker. Its fetching and eviction policy can be configured by users, depending on the application. We implemented and experimented with two management policies: FIFO and queue-lookahead.

1) FIFO

Under FIFO GPU memory management policy, GPU Memory Manager keeps a list of the models in GPU memory, sorted by the time when the model was added to the cache. When eviction is required to host a new model, cached models that are not actively in use get evicted starting from the earliest model on the list and until enough cache space is freed to hold the new model. Global State Monitor disseminates a bitmap for the cache to other workers in the system.

2) Queue-Lookahead

The FIFO policy has the drawback that the evicted model may soon be required by a subsequent task, resulting in a high number of evictions and fetch operations. To account for this case, we implemented an alternative queue-lookahead mechanism. Since the *execution queue* contains the tasks that are scheduled to be executed on the worker, this information can be used to implement a smarter cache management policy.

Upon eviction, GPU Memory Manager looks ahead into the *execution queue*, examining some fixed number of future tasks and their required models, and giving higher priority to models that will be used sooner. Then it sorts the list of models in GPU memory based on the priority. If a model must be evicted to make room for the model of the current task, GPU Memory Manager evicts models from the lowest priority to the highest. This way Navigator can avoid evicting models that are needed for the near-future tasks, and the associated overhead of fetching these models soon again.

D. Simulation

Although Navigator is a fully working system, we wanted a way to explore scenarios beyond what we can reasonably deploy on our limited-scale testbed. Accordingly, we implemented an event-driven simulator [git repo]. The simulator itself is similar to that of Sparrow [15]. It models all elements of the environment described in Section III, and supports the same request types as the real system, as illustrated in Figure 1. When estimating network delays and execution times, the simulator uses values we measured in the real system for input/output sizes, the sizes of AI models for our workload, and the task duration. Navigator's simulator models the task arrival, task waiting on the queue, task execution, and task dispatch as events, processed in the order of the simulated time. We validated the simulator against the real experiment using the same scale of 5 workers and the same workload, and observed that the performance difference between simulation and real experiment lies within 5% of the median numeric values.

VI. Experiments and Evaluation

We evaluate Navigator on a dedicated cluster, using 5 servers as the worker nodes and one as a client node. All workers are Dell PowerEdge R740 machines with dual Intel Xeon Gold 6242 processors and 192 GB memory. Each is additionally equipped with a Tesla T4 GPU having 16GB memory. The servers are connected by InfiniBand with RDMA-enabled

100Gbps network. Experiments in Section VI-B and VI-C2 are evaluated on a real system. Experiment in Section VI-D is performed using the simulator.

Our experiments focus on performance when scheduling multiple *DFG*s on a shared set of workers, and use a mix of the four workflows from Figure 1. On an idle system with AI models cached in GPU, the average completion times would range from 1 to 3 seconds, reflecting a dependency on relatively large models. In the experiment, the client node initiates a mix of concurrent requests. Text inputs to the translation 1a workflow and dialogue workflow 1c are randomly selected from the GLUE benchmark dataset [32]. The image inputs for image reading pipeline 1b and 3d perception pipeline 1d are sampled from the COCO dataset [16].

A. Performance Evaluation Metrics

The metric slow down factor measures how close the execution of each job instance came to the (possibly unachievable) lower bound for end-to-end latency. End-to-end latency is computed from when the job instance arrives at the system to the time when the last task in the job instance completes (in our examples, always shown as the "exit task"). Although the general problem of DAG-structured job scheduling is NP-complete, as mentioned in Section II-C, we can easily compute the lower bound for a given job instance: it is simply the time it takes to run the job instance with the maximum possible task parallelism and all inputs cached on GPU, and can be calculated from the *DFG* by assuming zero data transfer delay. The slow down factor of a job instance j is represented as

$$slow\_down\_factor_j = \frac{end\_to\_end\_latency_j}{lower\_bound_j} \geq 1$$

In reality, schedulers may never be able to achieve the slow down factor of one. It is nevertheless a useful metric for comparing the performance of different schedulers.

B. Scheduling Algorithm Analysis

Two primary considerations arise when scheduling workflows with *DFG* dependencies. One entails optimizing for dependencies within each job instance. For example, tasks that could be processed in parallel can be sent to different workers to maximize the level of concurrency during execution; tasks with large intermediate data dependencies can be grouped to run on a single worker to avoid data transfer delays. The other entails optimization reflecting the current state of the system as a whole, as represented in the shared metadata table. Here, decisions will avoid overloading a worker that is already at its peak capacity. Both approaches offer potential benefits, but they lead to very different schedules. The Navigator algorithm takes both factors into consideration, using a hybrid scheduling scheme. Below in Section VI-B1 we describe two scheduling schemes that each focus on one factor. Then Section VI-B2 compares Navigator with these two baseline options.

1) Baseline Schemes

JIT: A Just-in-time (JIT) scheduler makes individual task assignment decision as each task is about to be executed. Each time a scheduling action is needed, a JIT algorithm selects a task that has all prerequisite inputs available and assigns it to the worker that offers the earliest start time, obtaining the start time estimates by taking worker-state information from Global State Monitor and computing the worker start time using the worker wait time, model fetch time and intermediate data transfer time (if a required input would need to be moved from a different worker). The JIT optimization minimizes the finish time for each individual task.

HEFT: The HEFT algorithm introduced in Section IV-B optimizes for inter-job dependencies. Similarly to Algorithm 1, it first sorts tasks in the job instance, then assigns tasks to workers in the descending order of priorities. The optimization considers task parallelism and data transfer between dependent tasks. Unlike the customized planning Algorithm 1, a standard HEFT algorithm does not consider the worker wait time and the task-dependent AI model locality. It also lacks a dynamic adjustment mechanism: When a job instance is first triggered, HEFT would assign all the associated tasks to workers, and the workers subsequently adhere to the schedule.

Hash: The algorithm balances the load by randomizing task assignment to workers by hashing the task name combined with the request identifier. Hash is very simple and offers a uniform distribution of tasks to workers, and it is commonly used for workflow scheduling and load balancing.

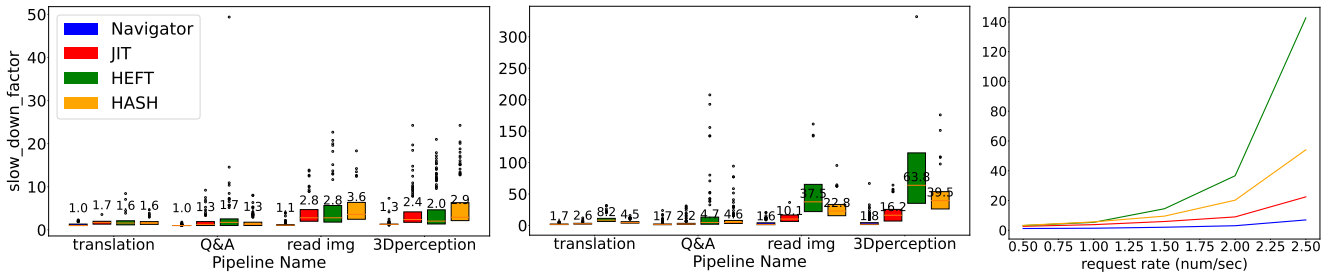
2) Performance Comparison Among Schedulers

Figure 6 shows results for an experiment that compares the scheduling schemes under steady low and high workloads, using mixes of the four job types presented in Figure 1.

Figure 6a shows the low-load case. Clients send pipeline processing requests at the average rate of 0.5 requests per second, with a Poison distribution on request types. The box plot shows the distribution of slow_down_factors for different instances, broken down by job category. The top and bottom of the box represent the first and third quartile of the data, respectively. The whiskers represent 1.5 times the upper and lower quartiles, and the dots show outliers. While all schedulers perform well under this case with close to optimal slow_down_factor, Navigator is the closest to the optimal schedule (slow_down_factor of 1.0).

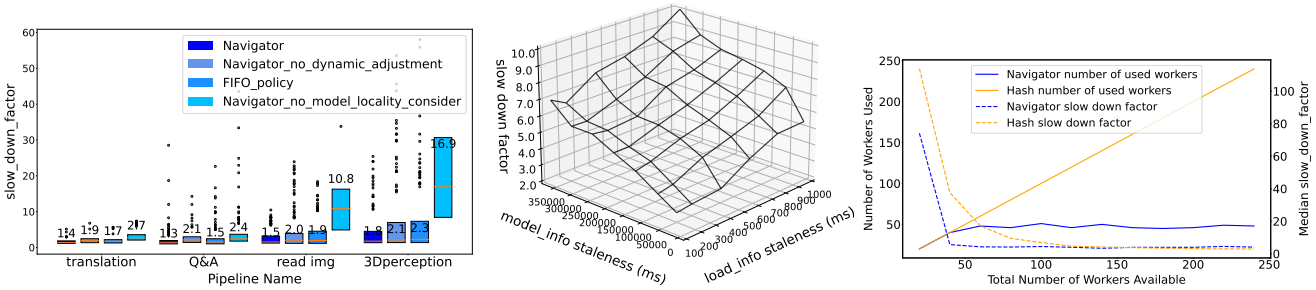
Figure 6b repeats the experiment with a job arrival rate having a mean of 2 requests per second under a Poison distribution. This places the cluster under more pressure, so the slow_down_factor for all four scheduling schemes increases due to the higher load. Navigator continues to outperform, yielding overall performance 2x to 4x faster than HEFT and Hash for the translation and Q&A pipelines, and 20x to 30x faster for the image description and 3D perception pipeline. The more extreme slow_down_factors in image description and 3D perception pipelines are because of their relatively short runtimes compared to translation and Q&A, which made these pipelines more susceptible to the system overheads caused by sub-optimal schedules.

Figure 6c experiments with different request rates under a Poison distribution. The plot shows the average



(a) Low-load scheduling performance (b) High-load scheduling performance (c) Varying load performance

Fig. 6: Comparison of scheduling schemes



(a) Ablation analysis findings (b) Varying information staleness (c) Simulation testing a larger scale

Fig. 7: Experiments showing the importance of algorithm features and the impact of data staleness on scheduling quality.

slow_down_factors for a mixed workload. Navigator has the closest to ideal performance at different request rates. The HEFT algorithm fails to adjust to the higher system loads due to not taking the workers' load into consideration. JIT performs better than the HEFT and Hash schemes due to its in-time task assignment mechanism, but not as well as Navigator because of it lacks a mechanism for intra-job coordination.

Table I shows average latency and other GPU-related performance metrics for the experiment presented in Figure 6b. GPU utilization is the percentage of time during which the GPU was actively processing computations. Memory utilization is the percentage of GPU memory being used, an indicator of memory-pressure on the GPU. Energy consumption is the total amount of energy consumed when running the full workloads. The GPU cache hit rate is the percentage of times when the required model is already in the GPU memory. The results indicate that Navigator consumes similar GPU resources and energy to other schedulers, yet achieves superior end-to-end latency.

TABLE I: Scheduler performance metrics

Scheduler	latency (s)	GPU utilization (%)	GPU memory utilization (%)	GPU energy use (J)	GPU cache hit rate (%)
Navigator	2.5	39	65	100760	99
JIT	5.0	42	68	103254	93
HEFT	18.0	38	73	103475	95
HASH	10.5	39	67	106851	91

C. Ablation Analysis

Navigator employs multiple features that contribute to scheduling. To understand the importance of each, we conduct an *ablation analysis* by selectively disabling features.

1) Importance of Scheduler Features

Dynamic task scheduling: To show the effect of using dynamic adjustment, this experiment disables dynamic adjustment that Navigator normally employs as each job instance executes, only retaining the preliminary planning phase. Figure 7a reveals a significant performance degradation.

Eviction policy: Next we investigate the importance of GPU cache-content awareness by selectively disabling the Memory Management policies introduced in Section V-C. As seen in Figure 7a, queue lookahead improves latency performance when the request rate is high, but has no significant impact at low request rates, where the initial task assignment decision is already close to optimal.

Model locality: The scheduler prioritizes workers whose GPU cache already contains the required models, as reflected in the metadata table. As shown in Figure 7a, Navigator incurs an 8x degradation without this mechanism. The GPU cache hit rate also decreases to 90% (as opposed to 99% when this optimization is enabled).

2) Sensitivity Analysis

Recall from Section III that we made a decision to limit the update rate for the SST-hosted metadata on which Navigator depends, forcing the system to run with potentially stale data.

The question arises of how sensitive Navigator's scheduling quality actually is as a function of the degree of staleness.

With this in mind, we set up a high-workload scenario and vary the frequency of updates using a "delay between update" configuration parameter, reflected in Figure 7b. In this plot, the x axis represents the staleness of the workers' load information as perceived by other workers, whereas the y axis represents the staleness of the workers' GPU cached model information as perceived by other workers. Observe first that the minimal slow_down_factor is achieved with the highest rate of SST pushes (10 per second), and the worst – with a very low rate (1 per second).

The graph tells us quite a bit more, however. Because AI model fetching to the GPU memory is an infrequent event, we see that Navigator can tolerate a much higher level of staleness for GPU cache-content information than for load. The delay for the latter has a significant impact beyond the threshold of 200ms (corresponding to 5 pushes per second). To make this concrete, at the 200ms rate in a cluster of 32 nodes, each participant would send and receive 160 RDMA updates per second (out of an RDMA rate limit of 75M/second): a trivially low overhead.

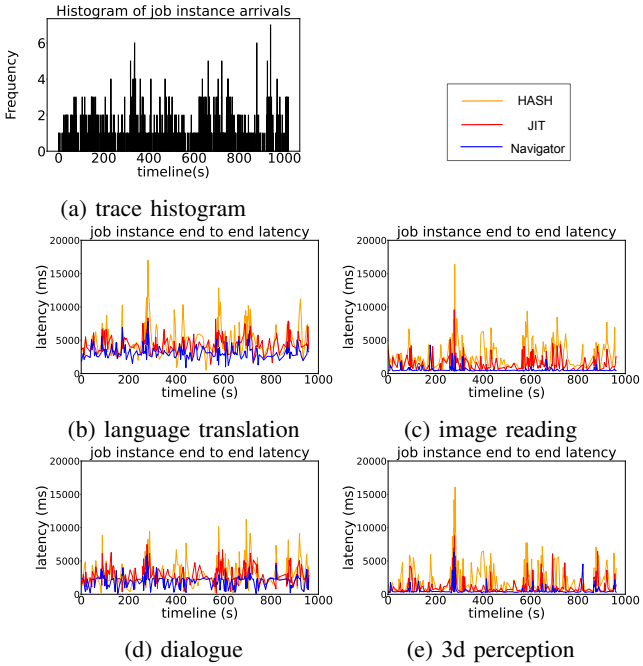
D. Scalability and Resource Consumption Analysis

Hashing is widely used for scalability in today's large clusters [45], [79], [28], under the assumption that it leads to a more even distribution of workloads and full use of resources. Our next experiment shows that in fact, Navigator uses hardware resources more efficiently. For this work we create a large-scale experiment using a simulation study. The workload is Poisson with a mean of 40 requests per second. We consider a range of up to 250 workers and examine a series of cluster utilization metrics as well as the median slow_down_factor over the job instances of all four types in Figure 1.

As seen in Figure 7c, Hashing is effective in the sense that it utilizes all available workers in each round, and the median slow_down_factor decreases as the number of workers increases, due to its inherent load-balancing property, approaching its lower bound at the scale of around 100 workers. Navigator prioritizes the workers with models in GPU memory and expands the worker set only if by doing so, mean job completion times would be improved. As observed in the experiment with the same workload, the slow_down_factor reaches its lower bound with just 50 active workers: half the resources required for the Hashing scheduler! These unused machines could be put into power-saving mode, reducing overall platform power consumption. With very large numbers of workers, Hashing slightly outperforms Navigator (at 150 machines and beyond), but to gain this tiny benefit it keeps three times as many machines active.

E. Production trace

Our next experiment considers scheduling performance for the workflows described in Figure 1 using a public trace from the Alibaba production GPU cluster [6], [67], [90]. We rescale this trace to adapt it to the capacity of our experimental system. Figure 8a shows request arrival rates on a timeline, while Figures 8b-8e show completion times as a function of arrival times. The plots make it clear that the Hash scheduling scheme is least tolerant of bursts of job instances, at least in a small worker cluster. Navigator has the best completion times even with these unpredictable request patterns.



(a) trace histogram

(b) language translation

(c) image reading

(d) dialogue

(e) 3d perception

Fig. 8: Experiment on production traces

VII. Related Work

The study of scheduling and cache management for AI has been addressed in several prior works. AI model optimization is the focus in Nimble [48] and Orca [71], which are orthogonal to the task scheduling optimizations in Navigator. Hardware management is the more central emphasis in [24], [52], [84], [66], [68], [49], [70], [67]. Clockwork [52] employs a fine-grained approach to manage the model in memory and during execution, and minimize tail latency. AlpaServe [84] utilizes model parallelism and statistical multiplexing, while the gpulet [66] scheduler takes the approach of spatio-temporal sharing of partitioned hardware resources to support heterogeneous AI models. Singularity [68], Antman [49], Walle [70] and PAI [67] schedule AI tasks in large-scale cloud production platforms. Cloud scalability encourages this type of elastic upscaling and downscaling, but these methods cannot directly transfer to small edge cluster with AI tasks that have large models and employ GPU accelerators. Navigator different from these systems in that not only consider the GPU resource sharing, but also the factor of multiple workflows that run on a shared edge cluster with resource constraints. Additionally, this work focuses primarily on workflows with just a single AI task per request, and generalization to *DFG* workflows would be a significant undertaking.

Scheduling for workflows with inter-job dependencies has been explored in many prior efforts [50], [18], [10], [51]. The estimation-based job planning and deferred correction mechanism used in Apollo [18] resembles the scheduling scheme in Navigator; while the task assignment algorithm in HEFT [10] resembles Navigator's job planning algorithm. However, direct application of these methods to the AI workflows in Navigator does not give latency-optimal scheduling result because they do not consider AI model re-use, GPU memory consumption and collocation of AI models.

Navigator's graphical workflows are similar to those considered in [23], [40], [20], [55], [78]. Stream Processing Engines (SPEs) scheduler [20] and Orion [78] use workload prediction model to estimate the worker waittime and provision the workers accordingly. These provisioning methods rely on workload prediction, but potentially sacrifice latency when the workload distribution drifts or if a burst of jobs puts the system under resource stress. InferLine[55] introduces a low-frequency combinatorial planner and a high-frequency auto-scaling tuner mechanism that it uses to upscale and downscale the number of workers as load varies. The process of worker selection for provision and up-scaling is less addressed in InferLine [55], which becomes nontrivial when the model sizes are significantly larger and when there are only limited GPU memories. Navigator addresses this scenario by scheduler and resource management co-design, which optimizes the task assignment while balancing the co-location of AI model objects in Navigator-managed GPU memory.

VIII. Conclusion

Navigator manages memory and schedules tasks in support of latency-sensitive AI workflows structured as DFGs. A novel scheduling-aware eviction policy yields high GPU cache hit rates, avoiding unnecessary model fetching and eviction. Our evaluation compared Navigator with state of the art schedulers under a variety of request rates and workload patterns, showing 2x to 6x speedup. Interestingly, although total resource use was similar, Navigator packed the work into fewer machines and left others idle, reducing energy consumption.

IX. Acknowledgments

We are grateful to Professor Luís Rodrigues for his many insightful comments and to Sagar Jha, who assisted in the early stages of this effort. Microsoft Corporation, Siemens Corporation, the Cornell Institute for Digital Agriculture, and the Research Council of Norway all contributed funding.

References

[1] Intel Xeon Gold 6242R Processor. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/gold.html.

[2] Intel Xeon Processor E5-2403. https://www.intel.com/content/www/us/en/products/details/processors/xeon/e.html.

[3] What is edge computing? https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-edge-computing,2024,Microsoft.

[4] AWS Edge Computing. https://aws.amazon.com/edge/.2024, AmazonWebServices,Inc.

[5] A Remote Direct Memory Access Protocol Specification. https://tools.ietf.org/html/rfc5040.

[6] Alibaba. Alibaba Production Cluster Trace Data. https://github.com/alibaba/clusterdata.

[7] NVIDIA Triton Inference Server Organization. Triton Inference Server. https://github.com/triton-inference-server.

[8] TensorFlow Serving. https://github.com/tensorflow/serving.

[9] J. D. Ullman. NP-Complete Scheduling Problems. In *Journal of Computer and System Sciences*, pages 384-393, 1975.

[10] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. In *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[11] Michael L. Pinedo. Scheduling: Theory, Algorithms, and Systems. *Springer Publishing Company*, 2008.

[12] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[13] Luiz F. Bittencourt, Rizos Sakellariou, and Edmundo R. M. Madeira. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[15] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[16] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft COCO: Common Objects in Context. In *European Conference on Computer Vision(ECCV)*, 2014.

[17] Christina Delimitrou, Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS*, 2014.

[18] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou, Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing, In *OSDI*, 2014.

[19] Zsofia K. Takacs, Elise K. Swart, and Adriana G. Bus. Benefits and Pitfalls of Multimedia and Interactive Features in Technology-Enhanced Storybooks: A Meta-Analysis. In *Review of Educational Research, 85(4), 698–739*, 2015.

[20] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic Stream Processing with Latency Guarantees. In *IEEE 35th International Conference on Distributed Computing Systems*, 2015.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen,A. Davis, J. Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, 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 *OSDI*, 2016.

[22] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *ACM SIGKDD KDD*, 2017.

[23] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*, 2017.

[24] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*, 2017.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*, 2017.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Conference on Neural Information Processing Systems*, 2017.

[27] Veton Këpuska and Gamal Bohouta. Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home). In *IEEE CCWC*, 2018.

[28] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX ATC*, 2018.

[29] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM*, 2018.

[30] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-End Speech Processing Toolkit. arXiv.org (March 2018). arXiv:1804.00015v1 [cs.CL].

[31] Ahmed Abbasi, Jingjing Li, Donald Adjeroh, Marie Abate, and Wanhong Zheng. Don't Mention It? Analyzing User-Generated Content Signals for Early Adverse Event Warnings. In *INFORMS Analytics Collections Vol. 16: Advances in Integrating AI and O.R.*, 2018.

[32] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. arXiv.org (September 2018). arXiv:1804.07461v3 [cs.CL].

[33] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, Andrė F. T. Martins, and Alexandra Birch. Marian: Fast Neural Machine Translation in C++. In *Association for Computational Linguistics(ACL)*, pages 116-121, 2018.

[34] Helsinki-NLP/opus-mt-en-fr. https://huggingface.co/Helsinki-NLP/opus-mt-en-fr. Hugging Face.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL), Volume 1*, 2019.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *Technical Report. OpenAI*, 2019.

[38] Kostis Kaffes, Neeraja J. Yadwadkar, Christos Kozyrakis. Centralized Core-granular Scheduling for Serverless Functions. In *SoCC*, 2019.

[39] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, Kenneth P. Birman. Derecho: Fast State Machine Replication for Cloud Services. In *ACM Transactions on Computer Systems*, 2019.

[40] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, Ravi Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *SOSP*, pages 322–337, 2019.

[41] K024/mt5-zh-ja-en-trimmed. https://huggingface.co/K024/mt5-zh-ja-en-trimmed. Hugging Face.

[42] Colin Raffe, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. In *Journal of Machine Learning Research*, 2020

[43] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Neural Information Processing Systems (NeurIPS)*. 2020

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Association for Computational Linguistics (ACL)*. 2020.

[45] Rohit Ranjan, Ishan Singh Thakur, Gagangeet Singh Aujla, Neeraj Kumar, and Albert Y. Zomaya. Energy-Efficient Workflow Scheduling Using Container-Based Virtualization in Software-Defined Data Centers. *IEEE Transactions on Industrial Informatics*, 2020.

[46] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[47] https://mxnet.apache.org/versions/

[48] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning, In *NeurIPS*, 2020.

[49] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *OSDI*, 2020.

[50] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *OSDI*, 2020.

[51] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gábor Rétvári. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *NSDI*, 2020.

[52] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up, *OSDI*, 2020.

[53] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. *European Conference on Computer Vision*, 2020.

[54] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *OSDI*, pages 599–616, 2020.

[55] Daniel Crankshaw, Gur E. Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph E Gonzalez, and Alexey Tumanov. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *ACM Symposium on Cloud Computing (SoCC)*, 2020.

[56] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[57] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, and Aditya Akella. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*, 2020.

[58] Sanaz Rabinia, Haydar Mehryar, Marco Brocanelli, and Daniel Grosu. Data Sharing-Aware Task Allocation in Edge Computing Systems. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2021.

[59] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *OSDI*, 2021.

[60] Robin Rombach1, Andreas Blattmann1, Dominik Lorenz1, Patrick Esser, Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021.

[61] Chao Wang, Kezhao Huang, Xuehai Qian. A Comprehensive Evaluation of RDMA-enabled Concurrency Control Protocols. arXiv.org (Jan 2021). arXiv:2002.12664v4 [cs.DC].

[62] Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. mT5: A massively multilingual pre-trained text-to-text transformer. In *Association for Computational Linguistics(NAACL)*. pages 483–498, 2021.

[63] Flink architecture: Tasks and operator chains. https: //bit.ly/3rTFplD. 2021.

[64] Ziwen Zhou, Tianming Zhao, Wei Li, and Albert Y. Zomaya. Distributed Online Resource Scheduling for Mobile Edge Servers. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2021.

[65] Dimitri Bertsekas, Robert Gallager. Data Networks: Second Edition. *Athena Scientific*, 2021. p. 149-271.

[66] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *USENIX ATC*, 2022.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *NSDI*, 2022.

[68] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou

Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv.org (February 2022). arXiv:2202.07848v2 [cs.DC].

[69] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *NSDI*, 2022.

[70] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, Tao Huang, Hui Shu, Jinde Song, Bin Zou, Peng Lan, and Guohuan Xu, Fei Wu, Shaojie Tang, Fan Wu and Guihai Chen. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In *OSDI*, 2022.

[71] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*, 2022.

[72] Weijia Song, Yuting Yang, Thompson Liu, Andrea Merlina, Thiago Garrett, Roman Vitenberg, Lorenzo Rosa, Aahil Awatramani, Zheng Wang, Ken Birman. Cascade: An Edge Computing Platform for Real-time Machine Intelligence. In *ApPLIED*, 2022.

[73] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, Ion Stoica. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *NSDI*, 2022.

[74] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a Visual Language Model for Few-Shot Learning. arXiv.org (November 2022). arXiv:2204.14198v2 [cs.CV].

[75] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzar. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. arXiv.org (Januaray 2022). arXiv:2201.11990v3 [cs.CL].

[76] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. arXiv.org (September 2022). arXiv:2204.02311v5 [cs.CL].

[77] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. arXiv.org (June 2022). arXiv:2205.01068v4 [cs.CL].

[78] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, Saurabh Bagchi. Orion and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *OSDI*, 2022.

[79] Kuljeet Kaur, Sahil Garg, Gagangeet Singh Aujla, Neeraj Kumar, and Albert Y. Zomaya. A Multi-Objective Optimization Scheme for Job Scheduling in Sustainable Cloud Data Centers. *IEEE Transactions on Cloud Computing*, 2022.

[80] Longlong Jing, Ruichi Yu, Henrik Kretzschmar, Kang Li, Charles R. Qi, Hang Zhao, Alper Ayvaci, Xu Chen1, Dillon Cower, Yingwei Li, Yurong You, Han Deng, Congcong Li, and Dragomir Anguelov. Depth Estimation Matters Most: Improving Per-Object Depth Estimation for Monocular 3D Detection and Tracking. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2022.

[81] Zihao Shao, Tonghua Su, Manyang Xu, Qinglin Liu, Ruipeng Han, and Zhongjie Wang. A Novel Heterogeneous Computing Middleware for Mobile AI Services. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2022.

[82] Doyeon Kim, Woonghyun Ka, Pyunghwan Ahn, Donggyu Joo, Sewhan Chun, and Junmo Kim. Global-Local Path Networks for Monocular Depth Estimation with Vertical CutDepth. arXiv.org (October 2022). arXiv:2201.07436v3 [cs.CV].

[83] Azure. Durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, Last retrieved: Auguest, 2023.

[84] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez and Ion Stoica. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *OSDI*, 2023.

[85] Attila Biró, Antonio Ignacio Cuesta-Vargas, Jaime Martín-Martín, László Szilágyi, and Sándor Miklós Szilágyi. Synthetized Multilanguage OCR Using CRNN and SVTR Models for Realtime Collaborative Tools. *Applied Sciences*, 2023.

[86] Krishna Murthy Jatavallabhula, Alihusein Kuwajerwala, Qiao Gu, Mohd Omama, Tao Chen, Shuang Li, Ganesh Iyer, Soroush Saryazdi, Nikhil Keetha, Ayush Tewari, Joshua B. Tenenbaum, Celso Miguel de Melo, Madhava Krishna, Liam Paull, Florian Shkurti, and Antonio Torralba. ConceptFusion: Open-set Multimodal 3D Mapping. arXiv.org (Feburary 2023). arXiv:2302.07241v2 [cs.CV].

[87] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Facev. arXiv.org (May 2023). arXiv:2303.17580v3 [cs.CL].

[88] vit-gpt2-image-captioning. https://huggingface.co/nlpconnect/vit-gpt2-image-captioning. Hugging Face.

[89] Krishna Murthy Jatavallabhula, Alihusein Kuwajerwala, Qiao Gu, Mohd Omama, Tao Chen, Shuang Li, Ganesh Iyer, Soroush Saryazdi, Nikhil Keetha, Ayush Tewari, Joshua B. Tenenbaum, Celso Miguel de Melo, Madhava Krishna, Liam Paull, Florian Shkurti, and Antonio Torralba. ConceptFusion: Open-set Multimodal 3D Mapping. arXiv.org (February 2023). arXiv:2302.07241v2 [cs.CV]

[90] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *Proc. ICPP*, 2023.

[91] Osamah I. Alqaisi, Ali Şaman Tosun, and Turgay Korkmaz. Containerized Computer Vision Applications on Edge Devices. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023.

[92] Jayden King and Young Choon Lee. eDashA: Edge-based Dash Cam Video Analytics. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023.

[93] Eduardo S. Gama, Natesha B V, Roger Immich, and Luiz F. Bittencourt. An Orchestrator Architecture for Multi-tier Edge/Cloud Video Streaming Services. In *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023.