HARNESS: Holistic Resource Management for Diversely Scaled Edge Cloud Systems

Ismet Dagli
Computer Science Department
Colorado School of Mines
Golden, CO, USA
ismetdagli@mines.edu

Justin Davis
Computer Science Department
Colorado School of Mines
Golden, CO, USA
jcdavis@mines.edu

Mehmet Esat Belviranli
Computer Science Department
Colorado School of Mines
Golden, CO, USA
belviranli@mines.edu

Abstract

Computing systems are evolving to be more ubiquitous, heterogeneous, and dynamic. Many emerging domains, such as Internet of Things (IoT), federated learning, and smart buildings, rely on a diverse edge-to-cloud continuum where the execution of applications spans various tiers of systems with significantly different computational capabilities. Computing resources in each tier, such as processing units inside of in-the-field edge devices and high-performance servers in datacenters, are handled in isolation due to scalability and resource segregation. This practice results in task mappings limited to only a subset of all available processing units, preventing an efficient overall utilization of the system.

In this paper, we propose a *holistic* approach to capture diverse computational characteristics of *edge-cloud* systems with arbitrary topologies and to efficiently manage computational resources with the whole continuum in the scope.

Our approach is built upon a *multi-layer* graph-based hardware (HW) representation and a *modular* performance modeling interface that can capture interactions and *interference* between computational resources in the system. We introduce an *orchestrator* mechanism that leverages the graph-based HW representation to hierarchically locate processing units to which a given set of tasks can be mapped while respecting the isolation between the computational tiers of an edge-cloud system. We demonstrate the utility of our approach on two distinct edge-cloud systems deployed in the field, improving the latency up to 47% over the best baseline with less than 2% scheduling overhead and reducing the average prediction error rate from 27.4% to 3.2%.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3729518

CCS Concepts

• **Computer systems organization → Cloud computing**; **System on a chip**; **Heterogeneous (hybrid) systems**.

Keywords

Edge-cloud computing, heterogeneous systems, task-based execution, resource management, system-on-chip

ACM Reference Format:
Ismet Dagli, Justin Davis, and Mehmet Esat Belviranli. 2025. HARNESS: Holistic Resource Management for Diversely Scaled Edge Cloud Systems. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3721145.3729518

1 Introduction

Computing in-the-field (*i.e.*, on the *edge*) is becoming more demanding. The increasing need for on-device intelligence results in a wider deployment of system on chips (SoC), embedding a variety of processing units (PU) and domain-specific accelerators (DSA), to efficiently run applications with minimal power and/or latency [69, 73]. Often, edge devices also rely on more powerful servers to store the data they collect [61], offload some of their computation [54] or synchronize with other devices [81]. In many emerging domains such as federated learning, autonomous systems, and AR/VR systems, the execution of the workloads spans numerous *nodes* with varying degrees of computational power (*i.e.*, heterogeneous), including *edge devices* and *servers* (or "cloud").

Over the last two decades, resource management for HPC systems [24, 57, 63] and data centers [67, 71, 72] has been broadly studied. In the meantime, edge platforms became more accelerator-rich, powerful, and efficient [38]. However, as the execution of emerging workloads evolved to span computing systems with vastly heterogeneous computational tiers, which we refer to as *diversely scaled edge-cloud systems* (DECSs), existing approaches fail to provide a *scalable resource modeling and management solution* that can *holistically* capture the *edge-cloud system*'s performance *accurately* and *adaptively* (see § 2 for details).

Specifically, DECSs have the following key characteristics which present unique challenges when considered together: (1) *High degree of computational heterogeneity*: Computational nodes (*e.g.*, edge devices and cloud servers) in a DECS constitute a multi-tiered computing topology. The nodes in each tier embed multiple PUs that have significantly diverse computational capabilities compared to the other nodes in the same and different tiers. (2) *Segregated/isolated computational environments*: Computational resources are segregated into various clusters to abstract or group nodes based on their functionality or isolation requirements (*e.g.*, mesh-IoT [40], function-as-a-service [45] and cloud gaming [12]). For example, the internal architecture and operation of the servers in the cloud are often invisible to edge devices and vice versa. (3) *Dynamically changing computational capabilities*: The performance of the system varies over time depending on the newly added or removed nodes and the slowdown caused by shared resource usages across various tiers.

In this paper, we propose *HARNESS*, **H**olistic and **A**daptive **R**esource ma**N**agement for diversely scaled heterogeneous **E**dge-Cloud **SyS**tems. *HARNESS* leverages a multi-layer *graph-based HW representation* to enable flexible and scalable abstractions of the computational resources in DECSs. The graph-based HW approach champions a modular strategy, allowing the incorporation of existing performance models, while still capturing interactions at higher levels across the system. *HARNESS* deploys a multi-tiered *Orchestrator* (*ORC*) mechanism to hierarchically locate resources a task could be mapped to, while taking the slowdown caused by shared resource usages at different tiers into account. *ORC*s enable scalable resource management across isolated/segregated computational node clusters without the need for any group to have complete performance models and task-assignment knowledge of the other group. *ORC*s utilize an internal *Traverser* mechanism to predict the shared resource slowdown for a given mapping between a set of tasks and target PUs. Our slowdown modeling approach uniquely decouples the standalone performance of a component and the slowdown caused by shared resource usage; hence, simplifying the performance modeling and increasing the prediction accuracy.

Overall, *HARNESS*, to the best of our knowledge, is the first framework to enable a *holistic resource management* approach utilizing *all computational resources* in heterogeneous edge-cloud-based systems while taking *multi-tier interference* and *dynamic HW topology changes* into account. Our work makes the following contributions:

- We present a *graph-based multi-layer HW representation* scheme that is capable of expressing *arbitrary topologies* of HW components and their *interactions* in DECSs.
- We devise a *Traverser* logic to automate the process of predicting the performance of a given set of tasks on a target set of PUs while also accounting for the *shared resource slowdown* among concurrent tasks.
- We design a multi-tiered, decentralized *Orchestrator* mechanism that scalably finds a mapping of a task to a local or remote PU while satisfying the task's constraints. Our proposed *ORC* mechanism uniquely supports *segregated resource clusters*, which is common in DECSs.
- We demonstrate the utility of *HARNESS* on DECSs from two different disciplines that we deploy in the field. Our experiments show a latency improvement of up to 47% and a reduction in the prediction error rate from 27.4% to 3.2% with less than 2% scheduling overhead.

2 Related Work

Performance modeling for diverse heterogeneity: Understanding and modeling the performance of DSAs in heterogeneous systems have been widely studied [4, 18, 19, 34, 35, 47, 48, 80, 82]. In these systems, interference factors that adversely affect performance include shared caches [6, 41], CPU sharing [16, 21, 51], GPU multi-tenancy [26, 33, 65], and multi-tasked DSAs [2, 36, 79]. However, all these works study a shallow, *i.e.*, *single-tier*, case of heterogeneity: They assume either there is a single type of DSA or a single flat layer of interaction between DSAs. In DECSs, however, computational nodes and the DSAs are connected in arbitrary topologies with multi-tiered hierarchies and they are often abstracted behind segregated clusters. *Existing approaches are not flexible and generalizable enough to capture multi-tiered interactions between diversely heterogeneous PUs in a DECS.*

Shared resource slowdown: Recent studies [17, 27, 53, 78] identified that tasks running concurrently on the PUs of shared-memory SoCs are subject to significant slowdowns. At the cloud level, multi-tenancy is often unavoidable [43, 52] since each server serves requests coming from multiple edge devices. These two types of slowdowns are often more severe at the performance-limited edge devices than more commonly studied high-performance systems in the cloud [6, 41]. *Therefore, if resource management mechanisms in DECSs do not account for slowdowns at different tiers of computation, predicted performance will be inaccurate, leading to missed QoS targets and oversubscribed resources.*

HW and system representation schemes: Graphs have been commonly used to represent system topologies in traditional large-scale systems [14, 20, 60], cloud providers [31], network-on-chip modeling [59, 83] and electronic design automation [46]. Among the most notable, *Hwloc* [11, 25, 42], an approach focused on HPC systems, uses a tree data structure to represent the underlying HW. However, when used for DECSs, these approaches (i) are *incapable of adaptive*

Table 1: Feature comparison against the state-of-the-art.

	ACE [74]	Borg [71, 72]	LaTS [81]	PARTIES [16]	ASPEN [66]	IRIS [34, 35]	CloudVR [49]	Hwloc [11]	EdgeMtrx[62]	***HARNESS***
Arbitrary hardware topologies	✓	✓	✗	✗	✗	✓	✗	✓	✗	✓
Scalable resource management	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
Not application-specific	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓
Shared resource slowdown	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓
Dynamic adaptability	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓
Heterogeneous PUs in a node	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓
Inter-node heterogeneity	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
Isolated resource management	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓

detection of shared resource contention and propagation of the slowdown across different layers, (ii) do not support segregated resource clusters and abstractions, and (iii) are not versatile enough to support dynamic changes to the HW topology.

Scalable and adaptive resource allocation and management: Many task-based execution frameworks [1, 9, 22–24, 57, 77] have been proposed for heterogeneous HPC systems. Interference-aware resource management within GPUs [76] and across large-scale clusters [24, 29, 30, 71, 72] is also widely studied for server-based systems [28, 70]. Most recently, multi-accelerator collaborative execution in embedded system SoCs [8, 10, 39, 68] attracted attention. A few studies [3, 7, 32, 62, 80] focused on task mapping in edge-cloud platforms, however, *they are either hand-tuned for specific application and HW, or they ignore diverse heterogeneity.*

Comparison of the features supported: Table 1 provides an overall comparison between *HARNESS* and other relevant works by the features they support: (i) the ability to represent arbitrary HW configurations and topologies within and across nodes, (ii) scalable resource management across nodes, (iii) support for arbitrary class of applications (*i.e.*, not application-specific), (iv) accounting for the slowdown caused by shared resource usage, (v) dynamically adapting to HW changes, (vi) modeling the performance of heterogeneous processors within a node, (vii) modeling diverse heterogeneity across the nodes, and (viii) supporting segregated/isolated resource clusters. Overall, *HARNESS* is the only work that supports all these features. Such comprehensive support is necessary to scalably and accurately model and manage resources in DECSs. The three studies (*LaTS* [81], *ACE* [74], *Multi-tier CloudVR* [49]) that we compare our work against are further explained in § 5.2.

3 Motivation

To demonstrate the need for a comprehensive resource manager for DECSs, we have developed and studied the

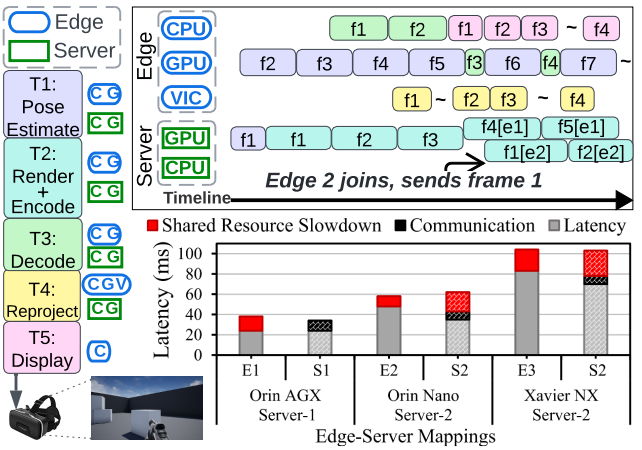

Figure 1: *Left:* The tasks in VR app and PUs they could run on. *Upper:* The frame pipeline for each PU. f4[e1] refers to frame 4 on edge device 1. *Lower:* [E]dge and [S]erver side latency, communication, and slowdown breakdown for VR application on a DECS containing 3 edge devices and 2 servers.

behavior of a real-life edge-cloud application. The application implements a remotely rendered 3D virtual reality (VR) environment, where users wear untethered VR glasses equipped with multi-accelerator SoCs. Most of the tasks on the VR glass (*i.e.*, edge device) could run locally, but a powerful remote server is still needed to deliver a low-latency and high-definition rendering of the 3D environment which VR users navigate using their body movements. To minimize end-to-end latency, we deploy an RNN-based predictive rendering approach [50] to speculatively predict the future poses of the VR user based on earlier body movements. We explore the execution of this VR application on a mini edge-cloud platform consisting of NVIDIA's Orin series edge SoCs and a remote server with a discrete GPU. In addition to CPU and GPU, Orin SoCs employ a special accelerator named Video Image Compositor (VIC), which can efficiently run some computer vision functions. The left part of Fig. 1 shows the linear order of five tasks of interest. Each task can be run on different set of PUs (GPU, VIC, CPU) on the server and edge devices. The upper-right part of Fig. 1 shows the frame execution pipeline when two edge devices share a single server for remote rendering operations. Initially, only one edge device (e1) is active with another edge device (e2) joining later during execution. Eight PUs (C, G, V for edge devices and C, G for server) work together to complete the five tasks for each frame captured by the edge devices.

We derive several key insights from Fig. 1: (i) Pose estimation is initially run on server GPU because it is faster, including the communication latency. However, since the server GPU is prioritized for rendering tasks, *T1:pose-estimation* (lilac) is swapped from the server to

the edge after the first frame. *This decision requires rapidly and scalably checking dynamic PU availability at every task assignment.* (ii) As new types of tasks start executing in the system, the mapping of older tasks could be changed depending on latency requirements. For example, the CPU of edge device one was executing *T3:decode* (green) tasks only, until *T5:display* (pink) tasks started arriving. The best decision once this occurs is to offload *T3:decode* tasks to the GPU, despite the GPU already handling *T1:pose-estimation*. *This decision requires heterogeneity, latency, and contention-aware mapping decisions.* (iii) New edge devices joining the system may cause a slowdown on the server due to multi-tenancy.For example, towards the end, two *T2:render* (turquoise) tasks begin arriving from both edge devices. The best mapping decision is to keep processing them on the server, despite the slowdown, since the latency requirements of the edge devices are still met. *This decision requires multi-tiered consideration of shared resource slowdown and adaptive task mapping w.r.t. the changing HW.*

Bringing the example one step further, suppose there are two slower edge devices (Orin Nano *E1* and Xavier NX *E2*) and one faster edge device (Orin AGX *E3*), all of which share two servers (*S1* and *S2*) for the remote rendering task. The bottom portion of Fig. 1 gives a breakdown of the time spent to process a single frame on each edge-server pair. While mapping the rendering tasks, the resource manager can identify that there are lenient latency constraints on the slower edge devices, since they will process other tasks in the pipeline slower. As a result, mapping their rendering tasks to the same server will not violate their latency requirements, although the collocated tasks will run longer due to shared resource use. On the other hand, this mapping will enable the other server to handle the stricter latency requirement of the faster edge device allowing the latency requirements for all rendering tasks from all edge devices to be met. The ability to make such a decision requires a comprehensive knowledge of all computational devices and task mappings in the entire system. However, due to resource segregation, neither edge devices nor the servers will have such knowledge. *This observation highlights the need for a holistic, multi-layer, contention-aware and adaptive resource manager that could operate under resource segregation.*

4 *HARNESS*: A Holistic Resource Manager for Diverse Edge-Cloud Systems

4.1 Design Requirements

To address the unique resource management needs of DECSs, we identify and target the following requirements while designing our proposed approach: (1) Computing systems with arbitrary and abstract topologies should be supported, (2) resource management should be de-centralized and support resource abstraction and segregation, (3) assignment of a task to a PU should be scalable and adaptive to dynamic changes in DECS, (4) the cumulative slowdown due to shared resource contention at different levels should be taken into account, and (5) modular integration of various performance and slowdown prediction models should be supported. Through the design of *HARNESS*, requirements (1) and (5) are captured by the graph-based representation (§ 4.3), requirements (2) and (3) are addressed by the *Orchestrators* (§ 4.5), and requirement (4) is satisfied by the *Traverser*s (§ 4.4).

4.2 Overview of *HARNESS*

HARNESS is composed of three major components:

- *HW-GRAPH*: A multi-layer graph-based HW representation that models the connections and interactions between the computational nodes and PUs in a DECS (§ 4.3).
- *Traverser*: A mechanism to automate the performance and slowdown prediction by traversing the tasks in a control flow graph (CFG) against the *HW-GRAPH* of a computational node (§ 4.4).
- *Orchestrators (ORCs)*: Hierarchically organized, per-node daemons that facilitate the assignment of tasks to PUs in a decentralized manner (§ 4.5).

An overview of *HARNESS* is depicted in Fig. 2. *HARNESS* fosters a decentralized and edge-triggered resource management scheme. Edge devices may run the same or different applications and these applications are assumed to be composed of tasks each corresponding to a unit of assignable computation (*e.g.*, kernel). The system developer/programmer, *i.e.*, the user of *HARNESS*, is expected to utilize the resource manager for the tasks representing computationally significant and acceleratable code blocks. Such tasks are to be labeled as `TASK`s (see *Application representation* paragraph in § 4.3) by the developer. Whenever there is a `TASK` or a set of `TASK`s that are ready to execute, the developer invokes our resource manager via the `MapTask()` API call. These `TASK`(s) are assumed to be either created directly or freed through dependency resolution. The developer passes the set of *TASK*s, any dependencies, per-task constraints, and the overall objective (such as minimizing the overall latency). The developer controls the granularity of the mapping by passing the number of desired `TASK`s to schedule, and *HARNESS* decides which node (*i.e.*, PU on device) to map each `TASK` onto. Overall, the steps *HARNESS* utilizes during operation are as follows: ① An *ORC* is associated with each higher-level node (*e.g.*, an edge device, a server or an abstracted/isolated node group) in the *HW-GRAPH* and run as a daemon. The `MapTask()` function is called on the local *ORC* to locate the resource(s) that the `TASK`(s) supplied by the developer can be mapped onto. ② The local *ORC* first

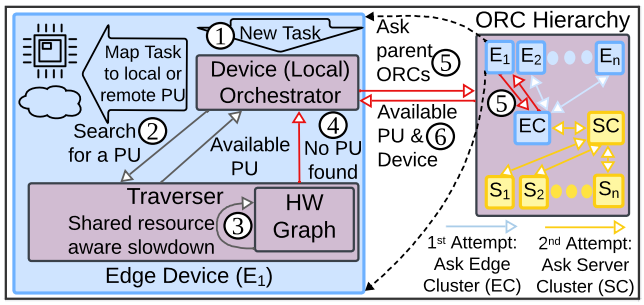

Figure 2: An overview of the *HARNESS* framework

invokes its internal *Traverser* to see whether any PU(s) in the same node could run the TASK(s) under given constraints. ③ *Traverser* utilizes the *HW-GRAPH* to query each PU in that device for its current TASK(s) and predicts the performance and potential slowdown if there are concurrently running TASK(s) sharing the same resources with that PU. ④ If the local device does not have a suitable PU that can execute the TASK under the given constraints, the local *ORC* requests resources from its parent *ORC*. ⑤ This triggers a hierarchical query across other *ORC*s—first among the edge cluster(s) and then, if needed, among the server cluster(s)—each invoking its respective *Traverser* to locate an appropriate PU. ⑥ Based on the returned resource/PU type (*e.g.*, CPU, GPU, VIC), the developer then invokes the proper implementation of the TASK to run on the designated resource/PU.

4.3 *HW-GRAPH*

The foundations of *HARNESS* are built upon a graph-based representation scheme, *HW-GRAPH*, customized to model the HW components and their complex interactions in a DECS. *HW-GRAPH* relies on a connected multi-layer graph topology [5, 37] to describe the hierarchical interactions between multiple levels of HW abstractions. In *HW-GRAPH*, a node corresponds to one of these: (i) a PU, such as a CPU core or GPU, (ii) a storage unit, such as cache or memory, (iii) a dedicated controller circuit, such as memory controller or network switch, (iv) an abstract component, whose internals are not known, or (v) a sub-graph representing a high-level component which groups smaller components, such as a CPU with multiple cores and caches or segregated resource clusters (*e.g.*, 'the cloud' composed of multiple servers). The edges in the *HW-GRAPH* correspond to interconnects linking the nodes listed above. Edges define (i) how PUs are connected to each other and the memory, and (ii) how higher-level nodes (i.e., edge devices or servers) are connected to each other in the network. The graph-based representation of the entire continuum enables *HARNESS* to *algorithmically* (*i.e.*, in a generalized and automated way) (a) traverse the PUs in an SoC or server, (b) locate the storage (*e.g.*, memory) and control components that two PUs share as they operate, (c) virtually group sets of computational devices (such as various edge or cloud clusters) for scalability, and (d) hierarchically identify other nodes in a DECS that a given node can offload its computation onto. Each node stores its own *HW-GRAPH* to represent its sub-components. All HW changes in the system (e.g., device additions/removals, network bandwidth changes) are captured via *HW-GRAPH* updates and propagated to the corresponding parent and child *ORC*s during execution. These features of *HW-GRAPH* enable seamless operation of *Traverser* and *Orchestrator* mechanisms, described later in §4.4 and §4.5, and let *HARNESS* support DECSs with any arbitrary computational and communicational topology.

Figure. 3 illustrates a *HW-GRAPH* representation of an edge device, NVIDIA AGX Xavier SoC, connected to a server in the cloud, similar to the VR application given in § 3. In this example, the edge device and server are the top-most layers in the graph (*i.e.*, layer 1), and they are connected via abstract components and links, which correspond to the unknown network infrastructure. The level of component detail increases in layers 2 and 3, and dashed connections across layers represent the relationship between abstracted and detailed versions of components. In the given example, *HW-GRAPH* can be used to automatically uncover a relation between DLA (deep learning accelerator) and PVA (programmable vision accelerator) residing under Layer 2 Vision Cluster. Concurrent execution on these DSAs results in the shared usage of multiple components: SRAM in `Vision Cluster` and LPDDR4x (*i.e.*, shared system memory). Intersection of the compute/memory paths used by two TASKs concurrently running on DLA and PVA algorithmically unveils these shared resources. While the example given in Fig. 3 is a highly detailed representation of the NVIDIA Xavier SoC, the developer may choose to represent only the relevant components of the system, such as the `CPU`, `GPU`, `DLA`, `PVA`, and the shared `LPDDR4x`. With these components, the developer could still model shared memory contention between the PUs, but the L2 cache based contention across the CPU cores would not be modeled if layer 3 is omitted.

4.3.1 The API:. The *HW-GRAPH* corresponding to a DECS system is created via an object-oriented interface. Every HW component derives from either `Node` and `Edge` objects. TASK(s) can only be mapped to higher-level `Node`s in the *HW-GRAPH*. For example, the `Convolution Engine` sub-component of the DLA in Fig. 3 cannot run a computational task alone, but the DLA can. Such higher-level `Node`s, *i.e.*, PUs, extends the `Predictable` interface and implement the `predict()` function so that the time it will take for the PU to run a specific task can be queried *algorithmically* by the Traverser (as explained in § 4.4).

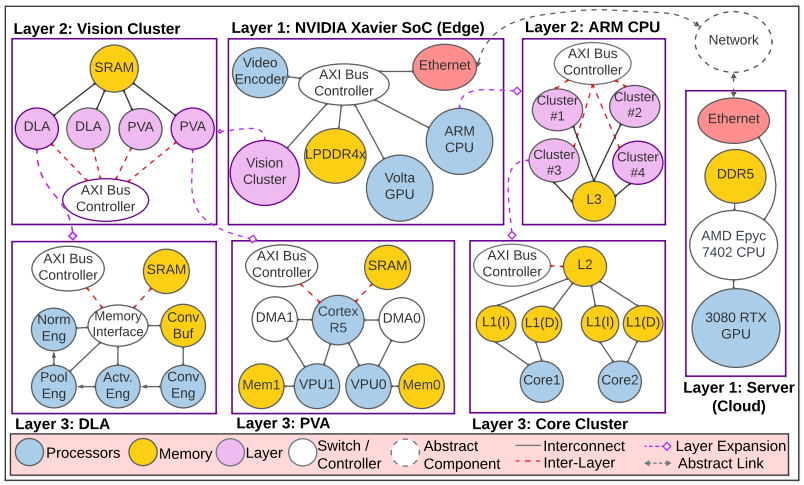

Figure 3: An example *HW-GRAPH* for a DECS made up of an NVIDIA Xavier SoC and a server with discrete GPU connected over a network.

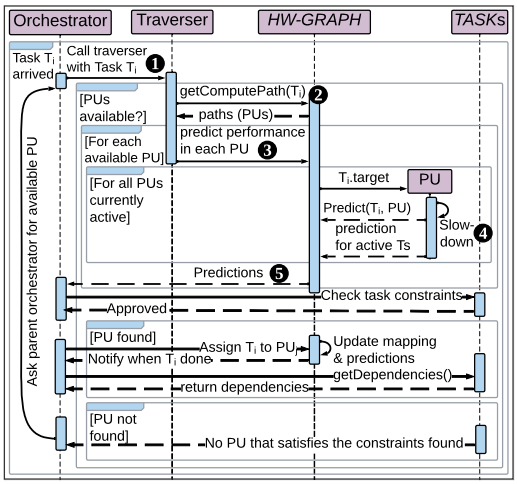

Figure 4: The internal operation of *Traverser* and how *Orchestrator*s utilize *Traverser*s.

4.3.2 Performance Prediction: The `predict()` function takes a `TASK` object as input to retrieve its previously modeled performance data for the PU-`TASK` pair. This function is designed in a *modular way* to support existing component-level performance prediction mechanisms, such as empirical profiling [64], Roofline [75], and analytical modeling [66]. To construct the *HW-GRAPH*, the developer creates `Node` and `Edge` objects that the target system is composed of. *HARNESS* ships with *HW-GRAPH* templates for commodity edge platforms and server components so that the developers can easily build upon. Automated creation of *HW-GRAPH* is left as future work. Each `Predictable` component implements `getComputePath()` function, which returns the shortest path between the PU and the other memory/control sources the `TASK` relies on. Such a list of resources is obtained during profiling and stored inside the `TASK` object. Traverser calls the `getComputePath()` function to automatically identify shared resources and account for any potential slowdown. In our experiments, we pre-profile the `TASK`s to be executed for all possible target PUs. While profiling `TASK`s beforehand may not be possible for a wider range of DECSs and applications, many DECSs run a pre-determined set of `TASK`s in a repetitive manner [13, 44]. Therefore, in this work, *we concentrate our efforts on resource management only* while also *enabling developers to modularly integrate their performance modeling methodology* via the `predict()` interface.

4.3.3 Application representation: We assume each device executes the same or different applications that are composed of computational regions (*e.g.*, kernels, API calls etc.) performing a specific function (*e.g.*, convolution, matrix multiplication etc.). Right before such function calls or code regions, the users of *HARNESS* (*i.e.*, application developers) are expected to create a `TASK` object that is composed of an identifier, input/output sizes, task constraints (*e.g.*, deadlines) and the list of local and remote PUs that the block could be executed on. *HARNESS* assumes that only computationally significant regions of the application will be denoted as `TASK`s and the remaining code will be executed locally. The user then invokes the *HARNESS* resource manager (explained in § 4.5) via the `MapTask()` API call and passes the `TASK` object as a parameter. At this point, the computational region corresponding to the `TASK` object is assumed to be ready to execute (*i.e.*, all dependencies are resolved) and will be mapped to the PU returned by the `MapTask()` call. If there are multiple `TASK`s ready to execute, then the developer may pass a set of `TASK`s, their dependencies (if any) and per-task constraints as parameters. The ability to pass one or more `TASK` objects at once enables the developer to control the granularity of the mapping – the trade-off between task mapping overhead and task granularity is evaluated in § 6.5. Once the `MapTask()` function returns a PU (local or remote) that satisfies the `TASK` constraints, the developer is expected to execute or offload the corresponding task region accordingly.

4.4 Traverser

We devise a *Traverser* as the mechanism to automate the process of predicting the performance of a given set of `TASK`s and their dependencies on a target set of PUs. *Traverser* acts as a *cost function* used by *ORC*s and it accounts for the potential shared resource slowdown among concurrently running `TASK`s while calculating the cost (*i.e.*, performance). *Traverser* uniquely automates this process by "traversing" the subcomponents of a higher level component in the *HW-GRAPH* at each task assignment. *Traverser* is invoked by the *ORC* (see § 4.5 for details) to predict the performance of a `TASK`

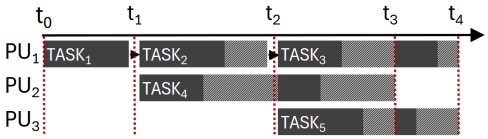

Figure 5: Timeline of five tasks completed on three PUs, with shaded areas showing additional slowdown and dashed lines marking *contention intervals*.

on a specific PU to check whether a mapping between them meets constraints without violating the constraints of existing tasks. *Traverser*, whose sequence diagram is depicted in Fig.4, operates as follows: ① Starting from the independent TASK(s) in the tasks set provided by the developer, it traverses through the dependencies in a time-ordered fashion by following the parallel & serial regions of the CFG and dependencies. ② Traverser honors the task-to-PU assignments, which are provided by *ORC*. ③ Traverser initially calls the `predict()` function to find standalone execution time on the PU in which a task is mapped. ④ Then, after identifying the contention intervals, as prescribed below, *Traverser* calls the `slowdown()` function with the collocated TASK information so that the slowdown is accounted for in the initially predicted performance and ⑥ returned to the *ORC*.

4.4.1 Contention intervals: When multiple TASKs run simultaneously on different PUs, TASKs will be slowed down non-uniformly throughout their execution depending on which other TASKs are running on the same computational node or PU at that particular time. Fig. 5 illustrates this behavior by depicting execution timelines of three workloads with five hypothetical TASKs that are running to completion on three PUs. To breakdown the slowdown calculation, we divide the initial predicted execution timeline (with no slowdowns) into *contention intervals*. Intervals are separated by time makers (*e.g.*, t_0 to t_4) each indicating the beginning or end of a task. This results in each interval to contain at most one task for each PU, hence simplifying the slowdown calculation. Then, the amount of slowdown for each interval is quantified iteratively as explained below.

4.4.2 Slowdown calculation: *HARNESS* uniquely decouples the calculation of slowdown from standalone performance models: (1) Only once for each system, the resources that can be shared are characterized and profiled for the slowdown they will experience per the amount of concurrent use they encounter. (2) Then, for a given resource, such as memory or a PU, each task is identified by the generalized amount of usage for that specific resource, such as requested memory throughput or core utilization, respectively. (3) Finally, during runtime, the `predict()` function uses concurrent TASKs' amount of usage for that specific resource,

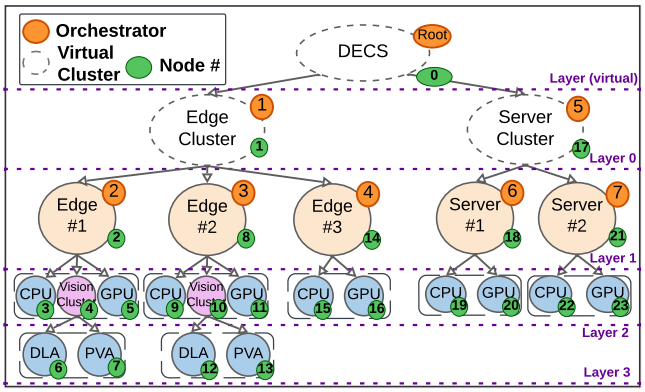

Figure 6: *ORC* hierarchy on a DECS with three edge devices and two servers.

and `slowdown()` function incorporates corresponding slowdown using the models built in the first step.

It is essential to note that the novelty of our slowdown calculation comes from the modular and decoupled integration methodology that scales across multiple tiers of DECS. The development of per-PU slowdown models is outside of the scope of this work, and existing slowdown models are utilized in our experiments.

4.4.3 Handling prediction inaccuracies: Since *HARNESS* relies on the performance models provided by the user, the predictions of execution times could be inaccurate. To accommodate predictions that are consistently off within margins, we integrate a rolling-window-based correction mechanism. *Traverser* achieves this by comparing the predicted execution times for the specific *TASK* and PU pair at hand with the actual time it takes to execute the *TASK*. If the difference is above a predetermined threshold θ, then *Traverser* calculates a correction factor σ by averaging the latest n predictions for the specific *TASK* and PU pair. Instead of relying solely on the predicted execution time τ calculated by the `predict()` and `slowdown()` functions, *Traverser* uses $\tau \times \sigma$ for upcoming task invocations. In our experiments, θ is set to 5%, which is a marginally higher value than the average model error rate that we measured in § 6.2. Also, we find that a window length of $n = 15$ predictions is sufficient to capture the dynamic changes in the system that affect prediction accuracy.

4.5 Orchestrator

The *ORC* mechanism is an integral component of *HARNESS* and facilitates assignments of tasks to PUs in a scalable way. To achieve this, *ORC*s work in a hierarchical tree topology and *ORC*s are responsible for finding an appropriate PU to map a given task to. An *ORC* daemon is created for every non-leaf computational node (*e.g.*, edge device or a server) of *HW-GRAPH*. They internally utilize *Traverser* and

HW-GRAPH to look for PUs in their hierarchy. Each *ORC* could only communicate with its parent and child *ORC*s, and none of the *ORC*s in the system is assumed to have the full knowledge of the *HW-GRAPH* or other *ORC*s in the system. Fig. 6 depicts an *ORC* hierarchy for the example DECS we described in § 3. There is an *ORC* associated with each edge device and server (2, 3, 4, 6, and 7). In addition, there are higher-level *ORC*s (1 and 5) associated with virtual abstractions created for edge and server clusters. There is also a *Root* level *ORC* that is only known by *ORC*s 1 and 5. There are no *ORC*s associated with the leaf-level nodes (*e.g.*, the PUs corresponding to node #3, #5, #6 and #7). This is because the parent *ORC*s (2, 3, 4, 6, and 7) are assumed to have full knowledge of the PUs that are immediate children of the node (*e.g.*, Node #2 corresponding to Edge #1) that they are overseeing.

Algorithm 1 Task Allocation via Orchestrator Mechanism

```
1:  Input: TaskId, Dependencies, Constraints, Objective,
2:  Output: targetNode
3:  Function MapTask(T_i, N_j, C_i, TD_i)
4:      for all Child ∈ ChildrenOf(N_j) do
5:          Add AskChildren(Child, T_i, C_i, TD_i) to BestNodes
6:      if BestNodes is not empty then
7:          BestNode ← select best in BestNodes on Objective
8:      else BestNode ← AskParent(N_j, T_i, C_i, TD_i)   ▷ Find new nodes
9:      return BestNode, Result
10: Function CheckTaskConstraints(T_i, N_j, C_i, TD_i)
11:     result ← InvokeTraverser(T_i, N_j, TD_i)   ▷ Predict slowdown for T_i
12:     if not SatisfyConstraints(N_j, T_i, C_i) then
13:         return result, False                   ▷ T_i's constraint is failed
14:     for all activeTask in N_j do   ▷ predict slowdown on active Tasks
15:         InvokeTraverser(activeTask, N_j, TD_i)
16:         if not SatisfyConstraints(N_j, activeTask, C_i) then
17:             return result, False          ▷ one active task's const. failed
18:     return result, True                            ▷ all constraints OK
19: Function AskChildren(N_j, T_i, C_i, TD_i)
20:     if IsLeaf(N_j) then
21:         if result ← CheckTaskConstraints(T_i, N_j, C_i, TD_i) then
22:             return result, N_j                     ▷ Task can be assigned
23:     else return MapTask(T_i, N_j, C_i, TD_i), N_j          ▷ Ask children
24: Function AskParent(N_j, T_i, C_i, TD_i)
25:     parent ← ParentOf(N_j)
26:     for all child ∈ ChildrenOf(parent) do   ▷ ask each child of parent
27:         if child ≠ N_j then
28:             return MapTask(T_i, child, C_i, TD_i), N_j
```

A pseudo algorithm for how *ORC*s work is given in Alg. 1. The working principles of the *ORC* mechanism are as follows: **(i)** For each new `TASK` T_i (along with its constraints C_i and dependencies TD_i), an edge device invokes the `MapTask()` function of its local *ORC* (line 3). **(ii)** The local *ORC* iterates over its children (line 19): (a) If the child is a leaf node N_j (*i.e.*, a PU that a `TASK` could be directly assigned) (line 20), then the *ORC* invokes *Traverser* on the PU to get a performance prediction that also accounts for the slowdown in the new `TASK` (line 11) and actively running `TASK`s (line 15). (b) If the child is an *ORC*, `MapTask()` is recursively invoked on the child *ORC* (line 23). (c) If a child PU that satisfies the `TASK`'s constraints is found, then that PU is returned. (line 7). **(iii)** Otherwise, the search is propagated to the parent *ORC* (line 8): (a) Parent *ORC* invokes `MapTask()` for the siblings of the local *ORC* (line 28). (b) If a suitable PU is not found, the search is propagated to other *ORC*s in a depth first search order.

To determine whether a suitable remote PU is found, the latency to communicate with such PUs from the origin PU is also factored in while checking for the constraints. To prevent deadlocks, when a remote *ORC* finds a suitable PU, that PU is provisionally reserved. The reservation is released if the originating *ORC* does not assign the `TASK` or timeouts. Once `MapTask()` returns, the user initiates the task execution via `ExecuteTask()` function. For local PUs, the task is executed locally by passing the input to the previously compiled binary directly. For PUs on other nodes, *HARNESS* internally invokes remote task execution.

Remote task executions are carried out similarly to the serverless computing paradigm outlined in [58]. `ExecuteTask()` serializes the *TASK*'s inputs and transfers the data to the remote node through the communication channel established between the local and remote ORCs. The binary required for the remote *TASK* and the PU pair is assumed to be previously initialized by the user via the `registerBinary()` function (see § 4.6 for details). Once remote execution is invoked, the remote ORC daemon service writes the input data to a predetermined directory expected by the *TASK* binary and initiates execution. Once the execution is complete, the output is streamed back to the local ORC initiating the *TASK* execution. For both remote and local *TASK* executions, the corresponding *TASK* binaries are executed directly on the remote system, without any containerization; as we solely focus on system utilization in this work and not the isolation of resources.

4.6 Usage of *HARNESS*

HARNESS framework provides the following:

- *HW-GRAPH* API for users to build HW-GRAPH models of their targeted compute nodes.
- Implementations for the ORC and Traverser mechanisms and interfaces for users to interact with local ORCs.
- Daemon services, communication and synchronization between ORCs at different compute nodes.
- Local and remote task execution mechanisms, including input/output data serialization and transfer for remote tasks (See the end of § 4.5 for details).
- A collection of *HW-GRAPH* models for commodity mobile and autonomous SoCs.

Users of *HARNESS* are expected to:

- Create the *HW-GRAPH* representation using the provided API calls or pre-made models shipped with *HARNESS*.
- Create *TASK* objects for the kernel executions that will be managed by *HARNESS*
- Declare `predict()` and `getComputePath()` functions for the PUs and *TASK*s that the PUs in the system could run.
- Provide and register device-specific kernel implementations, *i.e.*, binaries, for each TASK.
- Call the `MapTask()` function to find a target PU for a TASK and call `ExecuteTask()` to run the TASK.

Listing 1 provides code snippets for an example use of *HARNESS* from CloudVR application. The example shows the steps that the user needs to follow to add the *HW-GRAPH* for the Jetson Xavier AGX hardware, and then construct, map, and execute a "Reproject" task. First, the user adds a XavierSoC node to *HW-GRAPH*, indicating key components such as ARM architecture, accelerators (*i.e.*, GPU, DLA, PVA), and memory resources. Subnodes (ARMCluster, DLA_Cluster, PVA_Cluster, DRAM) in layer 2 are then added to represent additional details like caches, multiple accelerator cores, and shared memory. Next, a new `TASK` object is created with a latency constraint of 30 ms. Then, the user calls `registerBinary()` to let *HARNESS* know which binary a given *TASK* and PU pair should use. The user then implements `predict()` and `getComputePath()` functions for different PU targets that the *TASK* could run on. The steps described so far are performed only once for the underlying hardware (*i.e.*, *HW-GRAPH* related operations) or for each *TASK* type (*i.e.*, *TASK* related operations). During runtime, when it is time to launch the "Reproject" task in the application, the user calls `MapTask()` function to ask its local ORC for a suitable PU to execute the task at hand. If multiple tasks are asked to be mapped at once, then the user could specify an optional "dependency" parameter for the *ORC*s and *Traverser*s to take the dependencies into account when mapping the tasks. Finally the user calls the `ExecuteTask()` function for the `targetPU`, along with the input and output data locations. This function blocks until the execution of the task(s) finishes.

The *HARNESS* framework and the applications we use in this paper are available for download at https://github.com/hpsslab/harness.

Listing 1: Example usage of HARNESS APIs

```
import harness

hw_graph = harness.HWGraph.init()
orc = harness.ORC.init("ID", parent="Edge Cluster")

# An example of how one node is added in Layer 1
hw_graph.add_node("XavierSoC", harness.NodeSpec(arch="arm64", CPU="
      ARMCluster", GPU="VoltaArch", VisionCluster="2xDLA_2xPVA",
      VideoEncoder="H264", mem="DRAM"), parent="Edge Cluster")

# Layer 2 sub-nodes under Xavier SoC
hw_graph.add_node("ARMCPU", harness.NodeSpec(arch="arm64", clusters
      =4, mem="L3"), parent="XavierSoC")
hw_graph.add_node("VisionCluster", harness.NodeSpec(DLA_Cluster=2,
      PVA_Cluster=2, mem="SRAM"), parent="XavierSoC")
hw_graph.add_node("DRAM", harness.NodeSpec(type="LPDDR4x", memSize="
      8GB"), parent="XavierSoC")

# Interconnect each computation node in Layer 1 to SharedLPDDR4x
      with AXI edges
hw_graph.add_edge("ARM_Cluster, GPU, VisionCluster, DRAM,
      VideoEncoder", link="AXI", bandwidth="137GB/s")

# Create "Reproject" TASK object
task = harness.TASK(ID="Reproject", PU=["CPU","GPU","VIC"],
      constraint=[harness.LatencyDeadline("30ms")], inputSize,
      outputSize)

# Register pre-compiled binaries for the "Reproject" task
harness.registerBinary(task, "CPU", ".../bin/reproject_CPU")
harness.registerBinary(task, "GPU", ".../bin/reproject_GPU")
harness.registerBinary(task, "VIC", ".../bin/reproject_VIC")

# Implement the predict() function for PU-TASK pair.
def predict(task):
    profileData = harness.loadProfileinfo(f"/profiles/{task.ID}_{task
          .PU}.json")
    return {"latency": profileData.get("latency"), "resourceUsage":
          profileData.get("memUsage","cacheUsage")}

# Implement getComputePath() function for the GPU (implementations
      for other PUs are omitted)
def getComputePath(task):
    harness.Paths["Reproject"] = {
    "GPU": [XavierSoC, VoltaArch, DRAM] }
    return harness.Paths.get(task.ID)

# local ORC invokes mapTask (given in Algorithm 1) to find a target
      PU either locally or remotely
targetPU = orc.MapTask(task, constraint=task["Constraint"],
      dependency="Decode", objective="min_latency")

# Execute the task locally or remotely, as determined by targetPU
output = harness.ExecuteTask(task, input_data="/input/frames.bin",
      output="/output/result.bin", targetPU=targetPU)
```

5 Experimental Setup

5.1 Experimented Edge-cloud Applications

Application 1 (VR) Cloud-rendered VR: . This application is explained in § 3 and experimentation detail is given in § 5.2.

Application 2 (Mining) Smart Drill Bits: Underground mining requires operators to be close to the drilling machine to monitor conditions and the rock type being cut in order to prevent excessive damage to the drill bits and the mining machines. To minimize fatalities [15] and allow the operator to perform their role from a safer distance, an

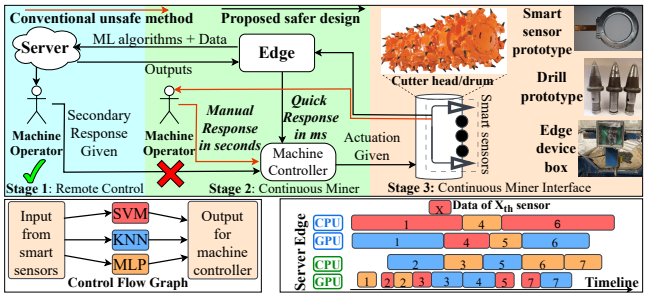

Figure 7: (Upper) The system operation for the Mining app. (Lower) Corresponding CFG and the pipeline of tasks.

in-the-field, edge-based real-time data analysis system has been developed in collaboration with Mining and Electrical engineering disciplines. In this application, we read the data in real-time through multiple smart sensors that are attached to the back of the drills. The experimental setup is depicted in the upper right section of Fig. 7. The application employs three machine learning (ML) tasks—support vector machine (SVM), k-nearest neighbor (KNN), and multi-layer perceptron (MLP)—to process smart sensor data. These tasks can be executed in parallel as shown in the lower section of Fig. 7. Since the cutter head drum, equipped with multiple smart sensors, rotates, the sensor data must be processed on edge devices and servers in real-time to identify the type of material being cut. If one of the ML algorithms detects an anomaly (*e.g.*, the rock type has changed), the system signals the machine controller to halt the operation.

5.2 System Configuration

5.2.1 Hardware: Table 2 lists the four edge devices with heterogeneous SoCs and the three servers with GPUs that we use in our experiments. The two diverse applications we tested *HARNESS* with are composed of numerous and different types of tasks. Fig. 8 lists the standalone, per-task execution times for each PU in the system, for both applications and also server-edge communication time for the VR app. Tasks in VR can target up to 3 PUs, which are CPU, GPU, and VIC, whereas ML tasks in Mining can run on the CPUs and GPUs of each server and edge device. The data arrival frequency (*i.e.*, injection rate) is the FPS value for each device in VR (*e.g.*, 30 FPS at 720p for Orin AGX) and 10 Hz per sensor in Mining. We utilize PyTorch for ML tasks in the Mining application. In edge devices, we use JetPack [56] 5.1.1 version and VPI [55]. Edges and servers are connected through WAN having a capacity of 10 Gb/s per device. Our slowdown model builds upon PCCS [78]. The novelty of our approach comes from (i) scalable integration of PCCS into the multi-tiered hierarchical *ORC* mechanism, and (ii) the decoupled approach that eliminates the need for pair-wise profiling of each task. On server GPUs, we estimate multi-tenancy-caused slowdown via profiling and empirical methodologies proposed by [26, 33]. We profile standalone execution times and slowdown characterization only once for each task/PU pair. Since the *HARNESS* enables decoupling the slowdown calculation, there is no need for pair-wise execution of potentially collocated tasks.

Table 2: List of targeted edge devices and servers.

Edge Devices:	Orin AGX	Xavier AGX	Orin Nano	Xavier NX
Server-1:	NVIDIA Titan RTX & AMD EPYC 7402			
Server-2:	NVIDIA GeForce RTX 3080 Ti & Intel i9-11900K			
Server-3:	AMD Ryzen 5800H & AMD Graphics			

5.2.2 Baselines: We compare against three studies:
LaTS [81] proposes a latency-aware task scheduling algorithm for real-time vision applications on heterogeneous edge-cloud systems. LaTS benchmarks the performance of system per task, periodically monitors the availability of PUs, and dynamically assigns the tasks based on the standalone execution time on PUs. However, LaTS does not utilize a shared resource contention mechanism.
ACE [74] constructs a unified platform for edge-cloud platforms considering high scalability. Yet, ACE is limited to static mappings only. So, ACE struggles to adapt to the changes throughout the execution and does not consider shared resource slowdown within a node.
Multi-tier CloudVR [49] specifically addresses the challenges associated with real-time remote VR rendering. Cloud-VR is adaptable to dynamically changing network conditions by balancing the computation and communication time by shrinking the frame resolution. However, CloudVR does not handle tasks other than rendering in the VR app.

The two baselines we compare against (*i.e.*, *ACE* [74] and *LaTS* [81]) are not shared-resource aware and ACE is also not capable of adapting to dynamic changes. So, the experiments in § 6.1 and § 6.2 comparing *HARNESS* against these baselines serve as an implicit ablation study for *HARNESS*.

6 Evaluation of *HARNESS*

In this section, we assess the utility of *HARNESS* with various experiments. For the first two experiments given in § 6.1 and § 6.2, we use VR and Mining apps, respectively. For other experiments, we interchangeably use both applications.

6.1 Overall Performance

In this main experiment, we assess *HARNESS*'s performance using a DECS comprising five edge devices and three servers (one from each with two Xavier NX). We define *latency* as the total time of a single VR frame spent on edge and server. Our goal is to minimize the pipelined latency of a frame in the VR app. Our results are reported in Fig. 9. *Overall, HARNESS improves pipeline latency from 11% to 47% over the*

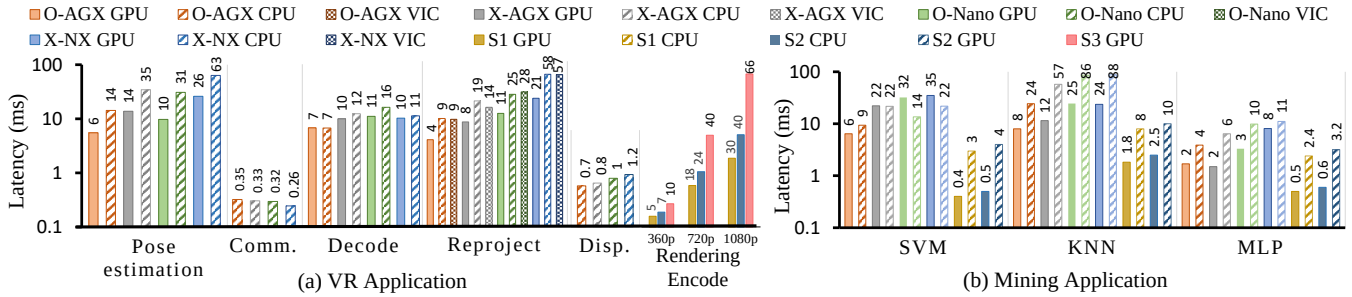

Figure 8: Standalone execution times of the tasks in VR and Mining apps for the various PUs of the edge devices and servers.

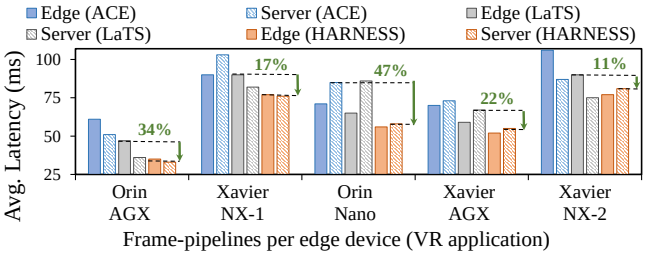

Figure 9: Average overall latency of *HARNESS* and others.

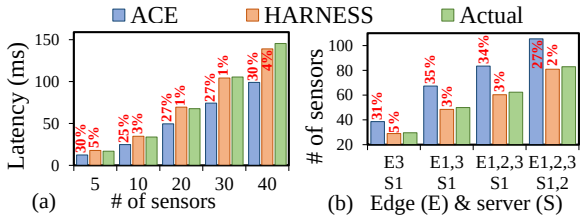

Figure 10: Prediction error rates of ACE & *HARNESS* for a system with (a) 1 edge device + server, (b) increased # devices.

best baseline. During the experiments, we observe that LaTS and ACE prioritize assigning *reproject* task to edge's CPU (if available) on Orin AGX, Xavier AGX, and Xavier NX over VIC since CPU's standalone time is superior over VIC. Yet, under the shared memory contention by multiple PUs, CPU generally performs worse than VIC since CPU shares the L4 cache with GPU and VIC has private buffers optimized for such tasks to minimize memory accesses. Additionally, LaTS and ACE choose to perform pose estimation on the edge devices' GPU and CPU, resulting in underutilized server resources and oversubscription of edge resources.

The average per-frame latency difference between all edge-server pairs is 11.8% for ACE, 12.6% for LaTS, and 2.4% for *HARNESS*, highlighting *HARNESS*'s adeptness at offering balanced system resource utilization. Among the pipelines of the five edge devices given in Fig. 9, the bottlenecks are on the server-side for the last three and on the edge-side for the first two. Given that servers are the bottleneck in three instances, we deduce that adding an extra server could enhance the performance of overall system.

6.2 Model Validation

We validate the performance prediction accuracy of *HARNESS* in the Mining app and compare it to a baseline model (ACE). In the first experiment, the objective is to determine the maximum number of smart sensor readings that both an edge device and a server (*e.g.*, Orin Nano and server 1 in this experiment) could process within a 100 ms latency threshold. We define *latency* in the Mining app as the time passed from the data being read by the sensor until all three ML tasks (*i.e.*, SVM, KNN, and MLP) are completed. This latency includes computation, slowdown, communication time, and the overhead spent to schedule the task. Fig. 10.a depicts the performance predictions made by ACE and *HARNESS* and compares them to the actual time it took to run. *Notably, HARNESS shows an average prediction error rate of 3.2%, significantly lower than 27.4% error rate of ACE.* A critical insight emerges for the experiments involving 30 and 40 sensors: ACE inaccurately predicts that tasks could be completed under the 100 ms latency threshold while *HARNESS* considers other factors, such as shared resource interference.

In the second experiment depicted in Fig. 10.b, we gradually increment the number of edge devices (Orin AGX-E1, Xavier AGX-E2, and Orin Nano-E3) and servers (server 1 and 2) in the system. Our goal is to determine an upper bound on the number of sensors that the DECS HW could handle under 100 ms latency. *HARNESS* can predict this with up to 98% accuracy whereas ACE overlooks the contention-related slowdowns and overloads slower edge devices, resulting in a falsely optimistic sensor count estimation.

6.3 Dynamic Adaptability

When network conditions degrade for an edge device, Multi-tier CloudVR [49] proposes decreasing the frame resolution at runtime to keep up the target FPS by reducing both computation (*e.g.*, some listed in Fig. 8) and communication time, whereas *HARNESS* can hierarchically update the scheduling assignments (*i.e.*, dynamically adapt) while also considering the delay introduced by the network communication since

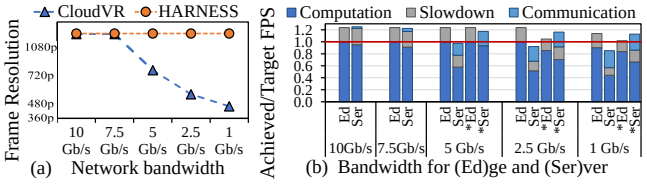

Figure 11: The variations in (a) video quality and (b) targeted FPS for changing network conditions. The Ed(ge) and Ser(ver) bars without * and with * denote the breakdown before and after *HARNESS* adapts to the change, respectively.

the *ORC* mechanism is triggered for every task assignment. We gradually decrease the network bandwidth capacity from 10 Gb/s to 1 Gb/s and analyze the change needed for the frame resolution to meet a fixed FPS target with the same experiment design in § 6.1. As demonstrated in Fig. 11.a, we observe that CloudVR targets lower frame resolutions after 7.5 Gb/s whereas *HARNESS* can keep up with FPS requirements by balancing the workloads through the entire system.

To gain a deeper insight into *HARNESS*'s ability to handle dynamic workload assignments, we further analyze the time breakdown of computation, slowdown, and communication on Orin AGX and target server(s). Fig. 11.b presents the ratio of the average achieved FPS over the targeted FPS. When bandwidth is reduced to 7.5 Gb/s, *HARNESS* successfully maintains the target FPS above the predefined threshold while still running mostly on server-2 as in the 10 Gb/s scenario. At a reduced bandwidth of 5 Gb/s, the *ORC* continues to assign the rendering task to server-2 for Orin AGX, but it avoids server-side GPU sharing. This alleviates the additional communication overhead on the server-side and maintains the target FPS for every edge device in the system since the rendering task on other edges can be assigned to different servers. In the most constrained scenario, at 1 Gb/s, *HARNESS* proactively identifies and assigns the rendering task to the best matching server (server-1) that can meet the target FPS requirements. Meanwhile, rendering tasks previously running on server-1 are reassigned to server-2.

When a new edge device joins an active edge-server system, a server must be assigned or shared to handle, at a minimum, the rendering and encoding tasks for the new

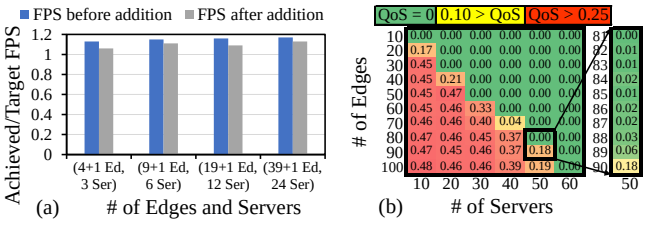

Figure 12: (a) *HARNESS* adaptability during edge device additions. (b) QoS violations as number of nodes scales.

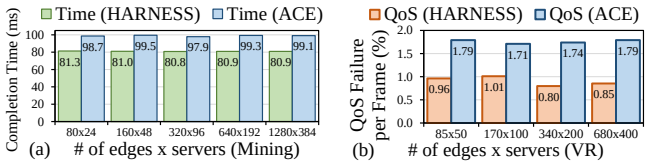

Figure 13: Weak scaling experiments for (a)Mining and (b)VR.

device. This necessitates the recalculation and rescheduling of multiple rendering tasks across servers. After the edge-device is connected, we dynamically add the device to our *HW-GRAPH* and the next time the *ORC* is called to map a task, it considers the new change. Fig. 12.a depicts how *HARNESS* maintains the desired FPS for varying server-edge counts. The blue bar denotes the worst FPS among edge-system pairs before the new device and the gray bar shows the FPS after *HARNESS* handles the workload changes.

To analyze the QoS failure for varying edge/server ratios, we gradually increase the number of edges and servers by 10 and measure QoS failure per frame (Fig. 12.b). We report the average QoS failure as the total number of frames violating the latency requirement over the frames completed. We observe that DECSs having more than or equal to a 2-to-1 edge/server ratio will result in noticeably high rate of failures since individual servers struggle to process more data than two edge devices provide.

While the experiments so far are the results of real-world testing, the experiments given in Fig. 12 and the subsequent experiments in the §6.4 rely on simulations that use the individual edge/server profiles validated in §6.2.

6.4 Scalability

Scaling experiments design: Weak scaling evaluates *HARNESS*'s performance while increasing the number of computing devices and keeping the average number of tasks per computing device constant. Strong scaling keeps the total number of tasks in the system fixed while the number of edge devices and servers is proportionally increased. In the Mining app, we measure the total completion time of the tasks which includes computation, data transfer, and communication between devices, slowdown per PU in each device, and scheduling overhead of *HARNESS* per task. In the VR app, as edge devices increase, the number of tasks also grows, as each device adds more frames and, consequently, more workload to be processed. The times reported for the VR app are comprised of similar contributors as with the Mining app. We compare the scalability of *HARNESS* against ACE [74], which we identify as the most relevant state-of-the-art study.

Weak scaling-1: For the Mining app, our initial setup includes 100 smart sensors with 80 edge devices and 24 servers (20 and 8 of each edge device and server listed in Table 2, respectively). Each experiment doubles the number of smart

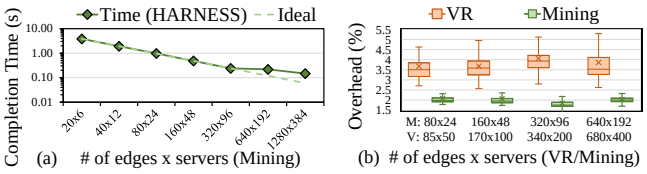

Figure 14: (a) Strong scaling experiments, (b) *ORC* overhead.

sensors, edge devices, and servers in the previous setup. Fig. 13.a reports the completion time per setup. *HARNESS* keeps the trend of completion time around 81ms as the number of devices and input sources proportionally increases whereas ACE generally keeps the completion time around 98ms due to the under-utilization of the system.

Weak scaling-2: For the VR app, we start with 85 edge devices with 50 servers and double them for each setup. Results shown in Fig. 13.b demonstrate that, for *HARNESS*, QoS failure rate is kept minimal as the system scales. Even though ACE has a similar trend in both applications, lack of contention handling and static assignment of tasks leads to under-utilization of the system, resulting in longer completion times and higher QoS failures.

Strong scaling: In the Mining app, we deploy concurrent 1250 sensors each of which triggers three ML tasks. Results reported in Fig. 14.a show a linear decrease in completion time up to the configuration of 640 edge devices. Beyond this point, the performance is principally constrained by the KNN task execution time on the Xavier NX edge devices, which emerge as the primary bottleneck.

6.5 *ORC* Overhead

We define the *ORC* overhead for a given task as the ratio of the initial `MapTask()` latency over the execution latency of the task. This time includes the computation time by the traverse and the time spent for communication and computation between local and remote *ORC*s until a PU is found.

DECS size: As the number of edges and servers are doubled, we measure the *ORC* overhead per task, average them to calculate the total overhead per iteration of both applications, and report the distribution in Fig. 14.b. The scheduling overhead is consistently preserved around 2% for Mining and 4% for VR apps. We observe that more than 90% of the overhead originates from communication between ORCs located on

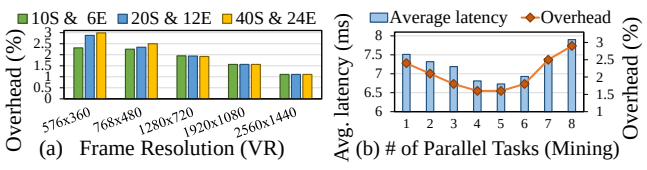

Figure 15: (a) Effect of task size over overhead for [E]dge & [S]erver pairs. (b) Granularity of parallel tasks assigned.

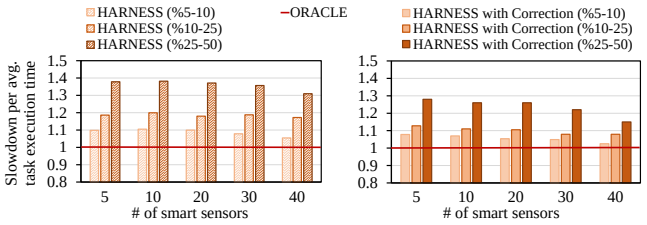

Figure 16: The slowdown in the average task execution times for varying amounts of artificial delays (between %X and %Y) added to task executions. Results in the left chart are produced with *HARNESS* version without the misprediction correction mechanism and the right chart is produced with the correction mechanism enabled.

different edge devices or servers. If ORC is local, there is no communication and the overhead of running *Traverser* on the local *HW-GRAPH* is minimal (around 0.1ms on average). If an ORC needs to communicate with remote ORCs over the network, every hop adds around 0.3ms, with our experiments having 2 hops max. Each ORC communication transmits a minimal amount of information over the network (*i.e.*, the parameters of the `MapTask()` function, as shown in § 4.6). It is important to note that profiling times are not included in the overhead calculations since profiling is performed only once per task-PU pair.

Task sizes: We investigate the effect of task sizes (*e.g.*, frame resolution size in the VR app) over the overhead of *HARNESS* by varying frame resolution while reducing the latency (*i.e.*, FPS) requirements per device proportionally. The results reported in Fig. 15.a show that, as the input size gets bigger, the number of tasks sent to servers per second lowers, thus decreasing the overhead of *HARNESS* per frame.

Mapping granularity: We investigate the effects of calling `MapTask()` with an increasing number of tasks and observe its relationship to the *ORC* overhead. Fig. 15.b shows the results for the Mining app. We observe that the average latency and overhead decreases until we assign 5 parallel tasks. However, mapping larger batches of tasks leads to load imbalance between computational nodes (especially in edge devices) because `MapTask()` is designed to assign a given set of tasks only to the PUs in the same computational node.

6.6 Handling Prediction Errors

We use empirical profiling in our evaluations; therefore, our prediction error rates are minimal, as shown in § 6.2. *Traverser* embeds a window-based correction mechanism to address larger and continuous deviations in performance predictions (see § 4.4 for details). To demonstrate the effectiveness of this correction mechanism, we conduct an experiment in which we artificially introduce delays in task executions to emulate performance mispredictions.

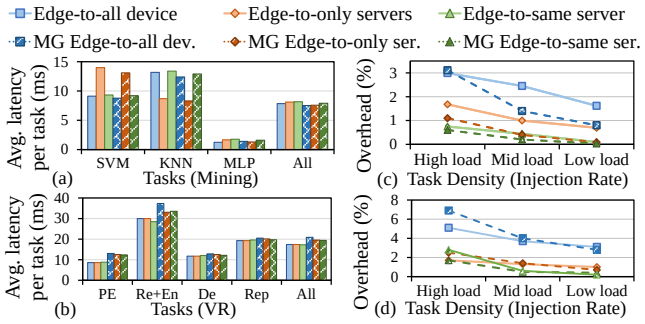

Figure 17: Latency and overhead for assignment strategies.

In this experiment, whose results are shown in Fig. 16, we run the Mining application continuously with five different sensor counts. We vary the artificial delay added in task executions between three intervals: 5%-10%, 10%-25% and 25%-50% of the original execution times. We report the resulting slowdown in the average task execution time compared to an Oracle that has perfect knowledge of mispredicted execution times. The graphs on the left and right side of Fig. 16 show how *HARNESS* performs without and with the misprediction correction mechanism, respectively. Our experiments show that the proposed correction mechanism in *HARNESS* is capable of mitigating the side effects of mispredictions to a considerable extent. As the added delay is higher, the resulting slowdown is less (compared to the delay to execution time ratio), due to the averaging of the predictions over a running window. In addition, we also observe that the negative effects of misprediction are less significant when the sensor count is increased. This is due to the increased amount of opportunities for overlapping present with higher sensors count.

6.7 Various Mapping Strategies

In the default mapping policy of *HARNESS*, as given in Alg. 1, local *ORC* checks child and parent *ORC*s hierarchically. Here, we explore alternate strategies. The first one (*i.e.*, Edge-to-only servers) involves direct communication from edge devices to servers, bypassing the communication between the *ORC*s of sibling edge devices. The second strategy (*i.e.*, Edge-to-same server) re-attempts to assign the task to the same node and PU that processed the task in the previous iteration. Lastly, we repeat the two strategies above by increasing the mapping granularity (MG) of ready tasks, similar to the previous experiment. Fig. 17.a and 17.b show average task latency for each strategy. In the VR app, we observe that first and second strategies improve system latency. Since the rendering task is often mapped to servers, these two strategies decrease the *ORC* overhead by skipping less powerful edge devices. In the Mining app, however, failing to query other edge devices leads to under-utilization of sibling edge devices and increases the latency. MG in the Mining app can improve the average latency whereas it does not in the VR app.

Fig. 17.c and 17d depict the *ORC* overhead when tasks are created in varying intervals for Mining (20 Hz, 10 Hz, and 5 Hz) and VR (1.10x, 1x, and 0.75x FPS of default values). We observe that frequent task creation has a higher overhead since the *ORC*s communicate more. In the VR app, grouping tasks generally causes higher overhead because some tasks cannot be mapped to a PU under current latency constraints and we end up splitting the tasks and rescheduling. This pattern is also observed in the Mining app under high load. Reprojection task is the only task that can run on VIC, yet during high load experiments, VIC may not successfully complete the task under stricter QoS requirements due to shared resource slowdown, which other baselines cannot spot. This results in using GPUs instead and leaving VIC idle. Please note that splitting is applied only when the user specifies a group of TASKs at once for the MapTask() API call. Single `TASK`s are not split into smaller pieces.

7 Conclusion

We propose *HARNESS*, a holistic resource management framework tailored for diversely heterogeneous edge-cloud systems. *HARNESS* uniquely takes the slowdown due to shared resource usage into account. We demonstrate the utility of *HARNESS* on two real-life applications from disparate disciplines deployed on the field, yielding 47% reduction in latency over baselines with less than 2% scheduling overhead.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-2124010 and CNS-2350228. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*. Springer, 863–874.

[2] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 940–953.

[3] Yogesh D Barve, Shashank Shekhar, Ajay Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. 2019. FECBench: A holistic interference-aware approach for application performance modeling. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 211–221.

[4] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor

architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–20.

[5] Mehmet E. Belviranli and Jeffrey S. Vetter. 2019. FLAME: Graph-based Hardware Representations for Rapid and Precise Performance Modeling. In *IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*.

[6] David Black-Schaffer, Nikos Nikoleris, Erik Hagersten, and David Eklov. 2013. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–10.

[7] Behzad Boroujerdian, Ying Jing, Devashree Tripathy, Amit Kumar, Lavanya Subramanian, Luke Yen, Vincent Lee, Vivek Venkatesan, Amit Jindal, Robert Shearer, and Vijay Janapa Reddi. 2023. FARSI: An early-stage design space exploration framework to tame the domain-specific system-on-chip complexity. *ACM Transactions on Embedded Computing Systems (TECS)* 22, 2 (2023), 1–35.

[8] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. 2021. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 159–172.

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[10] Halima Bouzidi, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, and Mohammad Abdullah Al Faruque. 2023. Map-and-conquer: Energy-efficient mapping of dynamic neural nets onto heterogeneous mpsocs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[11] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 180–186.

[12] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. 2016. A survey on cloud gaming: Future of computer games. *IEEE Access* 4 (2016), 7605–7620.

[13] Kun Cao, Shiyan Hu, Yang Shi, Armando Walter Colombo, Stamatis Karnouskos, and Xin Li. 2021. A survey on edge and edge-cloud computing assisted cyber-physical systems. *IEEE Transactions on Industrial Informatics* 17, 11 (2021), 7806–7819.

[14] Eddy Caron and Frédéric Desprez. 2006. Diet: A scalable toolbox to build network enabled servers on the grid. *The International Journal of High Performance Computing Applications* 20, 3 (2006), 335–352.

[15] CDC. 2023. NIOSH mine and mine worker charts. wwwn.cdc.gov/NIOSH-Mining/MMWC. (accessed on 08/16/2024).

[16] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 107–120.

[17] Ismet Dagli and Mehmet E Belviranli. 2024. Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'24)*. 243–256.

[18] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. 2022. Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1069–1074.

[19] Justin Davis and Mehmet E Belviranli. 2024. Context-aware Multi-Model Object Detection for Diversely Heterogeneous Compute Systems. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[20] R Kanniga Devi and G Murugaboopathi. 2019. An efficient clustering and load balancing of distributed cloud data centers using graph theory. *International Journal of Communication Systems* 32, 5 (2019), e3896.

[21] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. 2014. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 155–164.

[22] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193.

[23] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.

[24] Guangnan Feng, Dezun Dong, Shizhen Zhao, and Yutong Lu. 2023. GRAP: Group-level Resource Allocation Policy for Reconfigurable Dragonfly Network in HPC. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*. 437–449.

[25] Brice Goglin. 2014. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 74–81.

[26] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. 2015. dJay: Enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit GPU load balancing. In *Proceedings of the sixth ACM symposium on cloud computing (SoCC)*. 58–70.

[27] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 317–330.

[28] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. 2015. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations* 2, 3 (2015), 49–66.

[29] Nikhil Jain, Abhinav Bhatele, Louis H Howell, David Böhme, Ian Karlin, Edgar A León, Misbah Mubarak, Noah Wolfe, Todd Gamblin, and Matthew L Leininger. 2017. Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[30] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J Wright, and Laxmikant V Kale. 2014. Maximizing throughput on a dragonfly network. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 336–347.

[31] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whally, and Ed Snible. 2011. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *2011 IEEE international conference on services computing (SCC)*. IEEE, 72–79.

[32] Rathinaraja Jeyaraj, Anandkumar Balasubramaniam, Ajay Kumara MA, Nadra Guizani, and Anand Paul. 2023. Resource management in cloud and cloud-influenced technologies for internet of things applications. *Comput. Surveys* 55, 12 (2023), 1–37.

[33] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM*

Transactions on Architecture and Code Optimization (TACO) 16, 3 (2019), 1–25.

[34] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. 2021. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[35] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. 2024. IRIS: A Performance-Portable Framework for Cross-Platform Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[36] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 828–841.

[37] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. 2014. Multilayer networks. *Journal of complex networks* 2, 3 (2014), 203–271.

[38] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. 2023. Domain-specific architectures: Research problems and promising approaches. *ACM Transactions on Embedded Computing Systems (TECS)* 22, 2 (2023), 1–26.

[39] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 71–83.

[40] Huang-Chen Lee and Kai-Hsiang Ke. 2018. Monitoring of large-area IoT sensors using a LoRa wireless mesh network system: Design and evaluation. *IEEE Transactions on Instrumentation and Measurement* 67, 9 (2018), 2177–2187.

[41] Shin-Ying Lee and Carole-Jean Wu. 2017. Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 43–53.

[42] Edgar A León, Brice Goglin, and Andres Rubio Proaño. 2019. M&MMs: navigating complex memory spaces with hwloc. In *Proceedings of the International Symposium on Memory Systems*. 149–155.

[43] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*. 173–189.

[44] Hang Liu, Fahima Eldarrat, Hanen Alqahtani, Alex Reznik, Xavier De Foy, and Yanyong Zhang. 2017. Mobile edge cloud system: Architectures, challenges, and approaches. *IEEE Systems Journal* 12, 3 (2017), 2495–2508.

[45] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. 2017. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 162–169.

[46] Yuzhe Ma, Zhuolun He, Wei Li, Lu Zhang, and Bei Yu. 2020. Understanding graphs in EDA: From shallow to deep learning. In *Proceedings of the 2020 International Symposium on Physical Design*. 119–126.

[47] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. 2023. CEDR: A Compiler-Integrated, Extensible DSSoC Runtime. *ACM Transactions on Embedded Computing Systems (TECS)* 22, 2, Article 36 (3 2023), 34 pages. doi:10.1145/3529257

[48] Justin McGowen, Ismet Dagli, Neil T Dantam, and Mehmet E Belviranli. 2024. Scheduling for Cyber-Physical Systems with Heterogeneous Processing Units under Real-World Constraints. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 298–311.

[49] Abbas Mehrabi, Matti Siekkinen, Teemu Kämäräinen, and Antti Ylä-Jaaski. 2021. Multi-tier CloudVR: Leveraging edge computing in remote rendered virtual reality. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 17, 2 (2021), 1–24.

[50] Meta. 2019. Powered by AI: Oculus Insight. https://ai.meta.com/blog/powered-by-ai-oculus-insight/ (accessed on 08/16/2024).

[51] Amirhossein Mirhosseini and Thomas Wenisch. 2021. μSteal: a theory-backed framework for preemptive work and resource stealing in mixed-criticality microservices. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 102–114.

[52] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 579–596.

[53] Mohammad Alaul Haque Monil, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. 2020. MEPHESTO: Modeling Energy-Performance in Heterogeneous SoCs and Their Trade-Offs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. 413–425.

[54] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. 2021. Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 206–216.

[55] NVIDIA. 2024. VPI - Vision Programming Interface Documentation. https://docs.nvidia.com/vpi/ (accessed on 08/16/2024).

[56] NVIDIA. (accessed on 08/16/2024). JetPack SDK. https://developer.nvidia.com/embedded/jetpack

[57] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. 2021. A holistic view of memory utilization on HPC systems: Current and future trends. In *Proceedings of the International Symposium on Memory Systems*. 1–11.

[58] Tobias Pfandzelter, Aditya Dhakal, Eitan Frachtenberg, Sai Rahul Chalamalasetti, Darel Emmot, Ninad Hogade, Rolando Pablo Hong Enriquez, Gourav Rattihalli, David Bermbach, and Dejan Milojicic. 2023. Kernel-as-a-service: A serverless programming model for heterogeneous hardware accelerators. In *Proceedings of the 24th International Middleware Conference*. 192–206.

[59] Bharath Phanibhushana and Sandip Kundu. 2014. Network-on-chip design for heterogeneous multiprocessor system-on-chip. In *2014 IEEE computer society annual symposium on VLSI*. IEEE, 486–491.

[60] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. 2011. Multi-core and network aware MPI topology functions. In *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings 18*. Springer, 50–60.

[61] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. 2021. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems (FGCS)* 114 (2021), 259–271.

[62] Yuanming Ren, Shihao Shen, Yanli Ju, Xiaofei Wang, Wenyu Wang, and Victor CM Leung. 2022. Edgematrix: A resources redefined edge-cloud system for prioritized services. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 610–619.

[63] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. 2018. Scalable system scheduling for HPC and big data. *J. Parallel and Distrib. Comput.* 111 (2018), 76–92.

[64] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. 2021. Noise-Resilient Empirical Performance Modeling with Deep Neural Networks.

In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 23–34.

[65] Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. 2014. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 513–524.

[66] Kyle L. Spafford and Jeffrey S. Vetter. 2012. Aspen: A Domain Specific Language for Performance Modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE, IEEE Computer Society Press, Los Alamitos, CA, USA, Article 84, 11 pages.

[67] Xiang Sun, Nirwan Ansari, and Ruopeng Wang. 2016. Optimizing resource utilization of a data center. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2822–2846.

[68] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. 2023. Decentralized {Application-Level} adaptive scheduling for {Multi-Instance}{DNNs} on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 865–877.

[69] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Bin Ren, Yanzhi Wang, Shaoshan Liu, and Xipeng Shen. 2022. Brief industry paper: Enabling level-4 autonomous driving on a single 1K off-the-shelf card. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 297–300.

[70] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (2018), 1422–1434.

[71] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems (EuroSys)*. 1–14.

[72] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems (EuroSys)*. 1–17.

[73] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. 2018. *Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity*. Technical Report. USDOE Office of Science (SC), Washington, DC (United States).

[74] Luhui Wang, Cong Zhao, Shusen Yang, Xinyu Yang, and Julie McCann. 2023. ACE: Toward Application-Centric, Edge-Cloud, Collaborative Intelligence. *Commun. ACM* 66, 1 (2023), 62–73.

[75] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report 4. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States). 65–76 pages.

[76] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/2751205.2751213

[77] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. 2023. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 204–217.

[78] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. 2021. PCCS: Processor-Centric Contention-aware Slowdown Model for Heterogeneous System-on-Chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1282–1295.

[79] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. 2023. V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[80] Georgios Zacharopoulos, Adel Ejjeh, Ying Jing, En-Yu Yang, Tianyu Jia, Iulian Brumar, Jeremy Intan, Muhammad Huzaifa, Sarita Adve, Vikram Adve, Gu-Yeon Wei, and David Brooks. 2023. Trireme: Exploration of Hierarchical Multi-Level Parallelism for Hardware Acceleration. *ACM Transactions on Embedded Computing Systems (TECS)* 22, 3 (2023), 1–23.

[81] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. 2019. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1270–1278.

[82] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*. Association for Computing Machinery, New York, NY, USA, 859–873.

[83] Yong Zheng, Haigang Yang, Yi Shu, Yiping Jia, and Zhihong Huang. 2022. mTREE: A customized multicast-enabled tree-based network on chip for AI chips. *IEEE Embedded Systems Letters (ESL))* 14, 3 (2022), 143–146.