The Dissertation Committee for Agrim Bari
certifies that this is the approved version of the following dissertation:

Resource-Aware Scheduling and Caching for Heterogeneous Real-Time Workloads

Committee:

Gustavo de Veciana, Supervisor

Sanjay Shakkottai

Hyeji Kim

Sandeep Chinchali

Kerstin Johnsson

Resource-Aware Scheduling and Caching for Heterogeneous Real-Time Workloads

by

Agrim Bari

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2025

Dedicated to my mentors, family, and friends.

Acknowledgments

Since childhood, I have dreamt of becoming as learned as possible, and pursuing a Ph.D. felt like a natural step toward that vision. But dreams alone are not enough. This journey has only been possible because of the incredible people I met along the way—mentors, friends, and family who supported me through each high and low. What follows is a humble attempt to thank those who made this possible.

A Ph.D. is shaped most deeply by the advisor one works with. I feel extremely fortunate to have had Prof. Gustavo de Veciana as my advisor. In him, I found not just a mentor, but also a colleague and a friend. His unwavering support, deep patience, and constant encouragement helped me grow—both intellectually and personally. During times when I felt stuck, he would nudge me to dig deeper, to think more clearly, and to never settle. I fondly recall moments when I stumbled upon an idea/result over the weekend or on a holiday and could not wait to share it—and he was always eager to listen. For that curiosity, generosity, and belief in me, I will remain deeply grateful.

I am also thankful to the members of my dissertation committee: Prof. Hyeji Kim, Prof. Sandeep Chinchali, Prof. Sanjay Shakkottai, and Dr. Kerstin Johnsson. Their feedback and insight helped refine this work in meaningful

ways. I owe a special debt of gratitude to Dr. Kerstin Johnsson, with whom I collaborated on two projects. Our regular discussions taught me how to rigorously question assumptions, develop clarity of thought, and articulate my reasoning—skills I will carry with me far beyond this thesis. In many ways, she was a second advisor.

No research environment can thrive without the silent efforts of those who keep it running smoothly. I would like to thank the unsung heroes at WNCG—Karen Little, Melanie Gulick, Tom Atchity, Barry Levitch, Jaymie Udan, Apipol Piman, and Erin Salcher—who made sure that as students, we could focus entirely on our work. Your support behind the scenes did not go unnoticed.

Along the way, I found mentors in unexpected places. Ian P. Roberts, a dear friend and the best teaching assistant I have ever had, brought perspective and humor to this journey. His constant optimism helped reframe many difficult days. Another important influence was Love Babbar, a passionate teacher whose commitment to his students is inspiring. He helped me embrace programming not just as a skill, but as a powerful tool to explore ideas and solve problems. Through him, I learned that effective teaching can unlock a student's belief in themselves.

Graduate school can often feel lonely, but my peers at WNCG turned this into a time of camaraderie and shared purpose. To Saadallah Kassir, Advait Parulekar, Hasan Burhan Beytur, Satyam Kumar, Parikshit Hegde, Dennis Menn, Zach Clements, Georgios Smyrnis, Takeyun Lee, Juseong Park,

Eunsun Kim, Alperen Duru, Litu Rout, Manan Gupta, Kartik Patel, Ruichen Jiang, Ezgi Tekgül, Jean Abou Rahal, Geetha Chandrasekaran, Vinay Pai, Koushal Chagari, and Rajesh Sureddi—thank you. You made my time in a foreign country feel like home.

Each of you played a unique role: Saadallah, my research mentor and gym buddy, helped me navigate the early days of research. Advait, a fellow Leetcode enthusiast and gym partner, reminded me that Ph.D. students can write efficient code too. Hasan felt more like family than a friend. Satyam helped me stay connected to my undergraduate roots. Parikshit shared his infectious passion for research and GitHub tips. Dennis, with his constant probing, pushed me to question long-held beliefs. Zach brought laughter. Georgios reminded me of the power of good friendship. Takeyun, an expert coder and dedicated researcher, inspired discipline. And Juseong showed me that a Ph.D. student can also light up a dance floor. I am so thankful for each of you.

Beyond the university, I was lucky to have friends who stood by me like a rock: Aditi Mittal, Akshaya Shetty, Nayanika Ghosh, Tarannum Khan, Ashok Bhukkar, Shreya Dheenan, and many more. Our conversations—whether joyful, frustrating, or deeply personal—grounded me. You listened without judgment and brought light during difficult times.

Before this journey began, there were people who opened the doors that led me here. I am especially grateful to Prof. Shivendra Panwar for mentoring me during my research internship at NYU; to Prof. A.R. Harish,

whose undergraduate lectures sparked a lifelong interest in research; and to Prof. Y.N. Singh, who gave me my first opportunity to work on a real project. I also want to thank Shyara Parth, whose support made that internship at NYU possible.

And finally, my family—my true foundation. To my brother, Arindam Bari, whose words pushed me forward when I felt like giving up. To my mother, Minakshi Bhararia, who made countless sacrifices to see this day become real. To my father, Ramesh Chand Chaudhry, who taught me to think deeply and question boldly. And to my sister-in-law, Jaspreet Kaur Batra, who showed me how to fight for what one deserves. Thank you. This thesis is as much yours as it is mine.

Resource-Aware Scheduling and Caching for Heterogeneous Real-Time Workloads

Publication No. _____

Agrim Bari, PhD
The University of Texas at Austin, 2025

Supervisor: Gustavo de Veciana

Motivation and vision. There is growing demand for applications such as cloud robotics, Virtual Reality (VR), and Large Language Model (LLM)-powered chatbots that require low-latency responses and substantial compute and memory resources. These requests often originate from resource-constrained mobile devices and are offloaded to edge or cloud servers for execution. However, such servers are shared, resource-limited, and subject to unpredictable congestion. This thesis develops online algorithms to efficiently manage compute and memory resources on these servers, ensuring that service-level objectives (SLOs) are met under realistic workload conditions.

Computation offloading. We first address the challenge of offloading stochastic and heterogeneous compute jobs—such as those arising in cloud robotics—to edge or cloud servers under delay constraints and congestion on both the wireless link and the server. The goal is to maximize either the net

computational work offloaded or power savings at the client. We propose a measurement-based policy that combines two components: Probabilistic Admission Control and Cut Assignment (PACCA), which coordinates tradeoffs across diverse workloads, and Predictive Abandonment (PA), which drops jobs unlikely to meet their deadlines. Through extensive simulation across diverse loads and job profiles, we show that PACCA + PA significantly outperforms naive greedy policies and approaches near-optimal performance in many settings. We also demonstrate its robustness to uncertainty in job arrival rates.

Caching in VR. Next, we explore how to improve cache hit rates for data object requests at an edge server—for example, from clients navigating a shared VR environment. We focus on two aspects: a) how data objects are represented, such as through layered representation where each additional layer improves quality at the cost of more memory, and b) how to leverage patterns in client requests, such as groups of clients exploring the same VR space and making correlated requests.

We analytically show how layered representations impact cache performance under LRU, without relying on extensive simulation. Then, we introduce a measurement-based caching policy, Least Following and Recently Used (LFRU), which infers correlations in client requests to outperform traditional policies like LRU and LFU. To support this, we construct synthetic traces that emulate VR workloads with different correlation patterns. Our simulations demonstrate that leveraging such correlations, when present, can improve cache performance by up to 2.9× over LRU and 1.9× over LFU.

LLM inference scheduling. Finally, we consider the problem of serving LLM inference requests on a GPU. Each request goes through two stages: the prefill phase, which processes the entire prompt and generates the first token, and the decode phase, which generates the remaining tokens one by one. These phases place different demands on the GPU—prefill is compute-intensive, while decode is more memory- and bandwidth-sensitive. Moreover, the number of decode steps is not known in advance. The scheduling objective is to reduce Time To First Token (TTFT) during prefill while meeting Time Between Tokens (TBT) constraints during decode.

We propose a measurement-based scheduling policy that dynamically prioritizes decode requests based on real-time system conditions. This adaptive strategy allows the scheduler to better allocate GPU resources under changing load, outperforming static policies—especially when TBT constraints vary across requests. We validate our approach using experiments on an NVIDIA RTX ADA 6000 GPU, demonstrating lower TTFT while meeting TBT goals across a range of loads.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we lay the foundation for this dissertation by establishing the underlying research motivations. We begin by exploring the key features and inherent challenges associated with serving an emerging set of applications in Section 1.1. Subsequently, in Section 1.2, we provide an overview of the key contributions of this dissertation. Finally, in Section 1.3, we outline the organization of the dissertation.

1.1 Serving a New Generation of Applications

Latency-sensitive, resource-intensive mobile workloads—such as cloud robotics, immersive virtual/augmented/mixed reality (XR), and chatbots powered by Large Language Models (LLMs)—are now widespread, and their client base is growing rapidly. However, despite their diversity, these applications exhibit three common and tightly coupled properties:

- **Multi-resource footprint.** Each request can simultaneously saturate multiple resources—compute cores, high-bandwidth memory/cache, and communication resources—and different workloads stress these resources in markedly different proportions.

- **Stochastic arrivals.** Request streams fluctuate unpredictably with client behavior and exogenous events, yielding possibly bursty load profiles.

- **Stringent service objectives.** Tight deadlines or contractual Service-Level Objectives (SLOs) dictate client experience and provider revenue; violations of even a few hundred milliseconds are often unacceptable.

Running such workloads entirely on end devices would be ideal, yet the compact form-factor and thermal constraints of smartphones, XR headsets, and IoT devices impose hard limits on computation, memory, and energy budgets. Consequently, contemporary deployments are expected to rely on *Mobile-Edge Computing* (MEC) or cloud offloading, wherein resource-rich servers located at—or near—the radio access network execute the heavy computation and store data to serve requests. Offloading can reduce latency, curb on-device energy consumption, and unlock application footprints that would otherwise be infeasible. But offloading merely relocates the bottleneck: shared edge/cloud server resources are finite and can experience bursty congestion, so simply adding servers does not guarantee deadline compliance nor maximize system-wide utility. This dissertation tackles the above challenges through four progressively richer problem settings.

1.1.1 Managing Computation Offloading to MEC Under Hard Deadline Constraints

We first focus on using edge/cloud server as a *computing resource* to handle machine learning–powered, compute-intensive jobs—such as those in XR applications, autonomous navigation, or advanced photo editing. Mobile clients generate these jobs, each structured as a sequence of sub-tasks, modeled as a linear directed acyclic graph. These jobs can be partially or fully offloaded via the base station communication resources for execution on servers. However, this offloading must be managed under hard deadline constraints; a job is only useful if it finishes on time. We focus on addressing the following three challenges:

- stochastic job arrivals over limited shared computation and communication resources,
- strict deadline rewards (jobs yield value only if completed on time),
- and, heterogeneity in device capabilities and job profiles (deadlines, computation, and communication demands).

1.1.2 Caching Layered Data Objects

Next, in our second work, we study a setting where edge server acts as a cache close to the clients, storing data objects such as those used in Virtual Reality (VR) environments. Each object can be stored in multiple versions that trade-off size for quality, e.g., if a client is far from an object in a VR

environment, the system can serve a smaller, lower-quality version to save resources. Each data object version can be represented in:

- *Multiple Representation (MR)*: each version is a separate, independent data object.
- *Layered Representation (LR)*: higher-quality versions are formed by stacking additional layers on top of a base version.

In this work, we focus on the setting where client requests follow the Independent Reference Model (IRM), and address the following questions:

- Which representation-MR or LR-offers better cache performance?
- Can we develop analytical hit-rate models for common caching policies such as Least Recently Used (LRU) and Least Frequently Used (LFU) when using LR?
- How do the number and size of layers or versions in LR impact the cache's efficiency?

1.1.3 Caching Under Structured Request Correlations

In our third work, we move beyond the assumption of independent client requests. In many real-world systems, requests are not random or isolated—they often show structure and correlation due to shared client contexts or coordinated actions. For example, in collaborative VR environments, correlations may arise explicitly—such as students following a teacher through the

same virtual space—or implicitly, when many clients independently explore a popular location (e.g., New York), resulting in overlapping content requests. Our goal in this work is to explore the following questions:

- How well do traditional caching policies perform when client requests are correlated?
- Can we develop analytical models to understand caching policies under such correlations?
- Can we design caching policies that detect and leverage these request patterns to improve performance?

1.1.4 Service Level Objective-Aware Scheduling for Large Language Model Inference on Graphics Processing Units.

In our fourth work, we study how to manage the performance of Large Language Model (LLM) inference workloads—such as ChatGPT—running on a GPU. Each LLM inference request goes through two phases: the *prefill* phase, where the input prompt is processed and the first output token is generated, and the *decode* phase, where the model produces the remaining tokens one at a time in an autoregressive manner. These phases are governed by distinct latency SLOs. For the prefill phase, the key performance metric is Time To First Token (TTFT), which measures the time between the arrival of a request and the generation of its first output token. In contrast, the decode phase is evaluated using Time Between Tokens (TBT), defined as the delay between

the generation of consecutive output tokens. Our work addresses the following challenges:

- prefill and decode phases of requests stress different resources, so joint management of these is non-trivial,
- the length of a request's decode phase (total number of output tokens) is unknown at arrival which makes resource management subject to that uncertainty,
- and, TTFT and TBT SLOs pull in opposite directions: prioritizing prefills reduces TTFT but can starve decoding; prioritizing decodes protects TBT but may leave compute cores idling and increases TTFT.

1.2 Summary of Contributions

The research questions addressed and contributions through out this dissertation are as follows:

> How can we design lightweight, online policies that maximize system-wide utility for jobs that concurrently require computation, communication, and memory—before any single resource becomes the bottleneck?

First, when the edge server is treated purely as a computation resource, we introduce *Probabilistic Admission Control and Cut Assignment* (PACCA),

an algorithm that runs at the base station and jointly decides which jobs to offload and if so, which parts to offload. PACCA smooths bursty load by "shaping" traffic *before* it reaches the server. At run time we perform real-time congestion management through *Predictive Abandonment* (PA); where clients abandon offloading if they predict that communication and computation delays will preclude on time completion. Together, PACCA and PA form a closed-loop control layer that is robust to estimation inaccuracies in system loads and delivers performance which is near optimal in some scenarios.

Second, when the server is used for caching data objects, we propose ways to exploit the statistical structure in data object requests. For independent client requests, we show that the choice between the use of data objects with *multi-representations* (MR) and *layered representations* (LR) is driven by a tradeoff between storage overhead of LR over MR and request diversity. We derive a working-set approximation that predicts LRU hit probability under LR to within 1 % error, and empirically show that if more versions are available for a data object they need not be beneficial. We then generalize the classical Independent Reference Model by proposing the *Grouped Client Request Model*, which captures spatial and temporal correlations observed in VR traces. Under the Grouped Client Request Model, we empirically establish the sub-optimality of LFU for large caches and design *Least Following and Recently Used* (LFRU), a simple online policy that adapts to correlations in client requests. Extensive experiments on VR datasets show that LFRU increases hit ratio by up to 2.9× over LRU and 1.9× over LFU.

Finally, we tackle SLO-aware scheduling of LLM inference workloads on a GPU. Our focus is on settings where there is heterogeneity in the target TBT across requests. We design a practical scheduler that supports heterogeneous TBT targets while reducing TTFT and we empirically demonstrate improvements over current state-of-the-art in terms of TTFT while meeting all SLOs.

Collectively, these contributions deliver algorithms, analyses, and provide datasets that advance multi-resource management for MEC/cloud server systems across computation, caching, and LLM inference.

1.3 Organization

The rest of the dissertation is organized as follows. In Chapter 2, we present the management of edge offloading for stochastic workloads with deadlines. In Chapter 3, we provide answers to fundamental questions related to caching data objects with layered representations. In Chapter 4, we move beyond the IRM and focus on structured correlations in clients' cache requests. Then in Chapter 5, we focus on scheduling LLM inference jobs on GPUs while trying to meet SLOs. Finally, we conclude the dissertation in Chapter 6.

Chapter 2

Managing Mobile Edge Computing Offloading with Deadlines

In this chapter,[1] we study how mobile devices with deadline-sensitive, compute-heavy jobs can exploit edge/cloud server compute offloading with possible wireless and server congestion. We introduce a Predictive Abandonment (PA) policy, where users opportunistically cut and offload jobs but abandon offloading if they predict that communication and computation delays will preclude on-time completion, and a lightweight edge-side Probabilistic Admission + Cut Assignment (PACCA) mechanism that coordinates those decisions across heterogeneous users. Together, PA and PACCA substantially boost net offloaded work and device energy savings, approaching an analytic upper bound even under imperfect load estimates.

[1] This chapter is based on the published work in [6**?**]:

- A. Bari, G. De Veciana, K. Johnsson and A. Pyattaev, "Managing Edge Offloading for Stochastic Workloads with Deadlines," *Proc. ACM MSWiM '23*, 99–108.
- A. Bari, G. De Veciana, K. Johnsson and A. Pyattaev, "Dynamic Offloading for Compute Adaptive Jobs," *IEEE CCNC 2023*, 131–139.

Agrim Bari led the formulation of the problem, the design of policies, execution of experiments, and the writing of the paper.

2.1 Introduction

Next-generation applications and the MEC fabric. A new generation of applications powered by machine learning, e.g., AR/VR/XR, autonomous navigation, and photo editors, is pushing the computational and energy limits of mobile devices. One way to overcome these limitations while addressing low latency and privacy requirements is for users to (partially) offload computationally intensive jobs to shared Mobile Edge Computing (MEC) resources. By combining mobile devices' sensing, communication, and computation capabilities with computation at nearby edge servers and/or more distant cloud servers, one can envision a computation-communication fabric that can cost-effectively address the most demanding mobile users' compute jobs.

Benefits of compute job offloading. There are several benefits mobile devices can reap from offloading. First, devices with insufficient computation resources may only be able to complete a job through offloading. Second, even if a device can complete a job, it may opt to offload to save energy and/or reduce its computational work to allow for computation of other jobs. Third, offloading a job might enable a mobile device to leverage powerful MEC/cloud computation resources to speed up job completion. In this chapter, we introduce policies that maximize the amount of work offloaded from devices, the amount of energy devices save while completing jobs on time, or the fraction of jobs that experience reduced execution times.

Managing compute job offloading. To realize these benefits, one must orchestrate offloading across various resources and account for the possible costs of doing so. In general, offloading a compute job may include the following steps: (i) (partially) computing the job on the device; (ii) transferring data to an edge/cloud server via a shared wireless link for performing the remaining computational work; (iii) performing further computation on the edge/cloud server; and (iv) transferring the results back to the device. These steps may involve shared computation resources on the mobile device, shared wireless channels, and shared edge/cloud computation resources, which may become congested under stochastic loads. Such systems also face significant heterogeneity in terms of devices' computation and/or communication capabilities as well as running heterogeneous compute jobs with different Quality-of-Service (QoS) requirements, e.g., constraints on completion time. To address these complex challenges, we present an offloading framework that combines an offload admission control policy with a lightweight user-driven offload abandonment policy.

DAG job cutting and offloading. In this work, we focus on offloading compute jobs that can be roughly modeled as linear Directed Acyclic Graphs (DAGs), where the nodes represent computational *sub-tasks*, e.g., *Deep Neural Network (DNN) layers*, and edges represent the *data dependencies* and potential cut locations between sub-tasks, e.g., see Fig. 2.1. In our work, we use the findings of [70] to only select a single cut location for time-sensitive jobs. The authors show that, under a block fading/Markovian stochastic chan-

nel, the optimal policy for offloading computational work from a local device to an edge/cloud server would at most offload once. This result assumes a congestion-free system, a device expends more power on processing a job than sending/receiving data, edge server processes faster, and flexibility to execute sub-tasks on either device or edge server. In the shared MEC fabric, a job's optimal cut location depends on its computation-communication requirements per cut location, completion deadline, current wireless network conditions, and the computational resources of the device it is generated on relative to available networked edge compute servers. It also depends on the operator's preferences [46] (e.g., rewards and/or fairness). Thus, some form of coordination of offloading decisions is necessary.

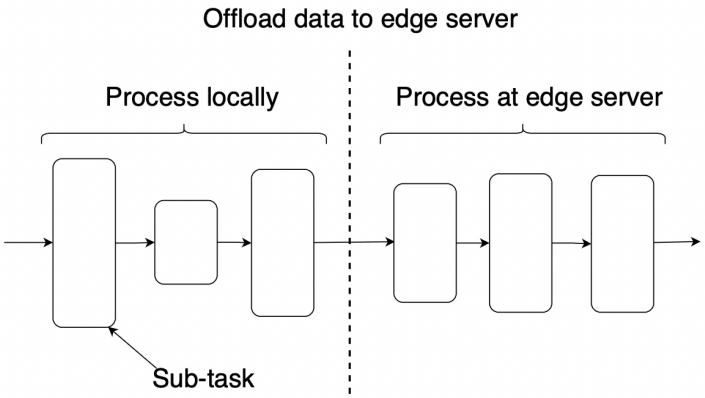


Figure 2.1: Cutting and offloading of a linear DAG.

Applicability of DAG model. There are several works [64, 36, 40, 31, 57] in the literature that embrace the linear DAG model as an effective abstraction/approximation of jobs that might particularly benefit from offloading. The underlying driver is the layered structure of DNNs, currently used in

several applications ranging from image classification, facial, digit, and speech recognition to many others. We believe that a substantial volume of future workloads will have structures like linear DAGs, where there is flexibility to cut and offload jobs. For a general DAG, the researchers [69] have extended their offloading policy for a linear DAG to a general DAG by exploiting the notion of a DAG's critical path.

2.1.1 Related work

Offloading problem. Offloading of compute jobs to MEC has been widely studied in the literature, which can be divided into two main categories: binary offloading and partial offloading. In binary offloading, a job is either executed on the device or offloaded to one or more edge servers for execution, intending to optimize performance metrics such as average computation delay or energy consumption. In most cases, the binary offloading problem is NP-hard, and various heuristics, approximation, and stochastic approaches have been proposed, see e.g., [13, 68, 3, 14, 75, 60, 48, 66]. Researchers in [3] explore the behavior of users when making decisions about offloading compute jobs in a multi-MEC server environment. They propose a Prospect Theory-based solution, considering users' risk-seeking or loss-aversion behavior. However, [3] has limitations, focusing on jobs consisting of independent sub-tasks and lacking a strict deadline constraint. Similarly, authors in [68] propose a game theoretic solution.

In [14], authors address fairness and maximum delay tolerance in hybrid

fog/cloud systems by jointly optimizing computation migration and resource allocation (including computing and bandwidth). They propose a suboptimal algorithm to solve the formulated mixed integer non-linear programming problem. Another paper, [75], focuses on joint computation offloading decisions, resource allocation, and content caching strategy. The authors transform the problem into a convex form and solve it in a distributed and efficient manner using optimization theory tools. Given a set of jobs and multiple edge servers, [13] proposes an approximate solution considering dynamic voltage frequency scaling for mobile devices. Their heuristic algorithm optimizes job offloading and frequency scaling decisions. However, all of the aforementioned works focus on static regimes where all jobs are assumed to be present at the beginning and ignore congestion on wireless channels and edge servers.

Partial offloading. In partial offloading, a job represented as a DAG, see Fig. 2.1, can be offloaded at several cut locations. Several research studies such as [70, 69, 10, 62, 44] have been conducted on partial offloading in the context of edge computing. In [70], the focus is on minimizing energy consumption while meeting latency constraints in a collaborative mobile device and edge server environment with stochastic channels. They propose a polynomial time algorithm for efficient job execution. Building on this work, [69] extends the approach to encompass general DAG frameworks beyond linear ones. In [10], the authors employ Reinforcement Learning (RL) to explore offloading multiple users' jobs to multiple servers. The users offload heterogeneous jobs over time-varying wireless channels. However, the RL-based policy

requires re-learning for each new environment (number of users, job profiles, channel capacity, etc.). To overcome this limitation, [62] introduces the use of Meta Reinforcement Learning, enabling the RL agent to quickly adapt to new environments without re-learning. [44] investigates an online offloading framework similar to ours, where heterogeneous job types with deadline constraints arrive in the network according to a stochastic process and are executed dynamically over time. The authors propose a heuristic approach by relaxing the deadline constraint. The objective is to minimize the average makespan time.

2.1.2 Contributions and organization

The main contributions of this chapter are summarized as follows:

- We tackle the design of offloading policies for stochastic and heterogeneous job requests (distinct deadlines, cut locations, and computation-communication requirements per cut locations) under strict deadline constraints.

- We introduce and evaluate three "revenue" models to capture the possible benefits of offloading. The first model, termed net timely offloaded workload, aims to maximize the amount of computational work offloaded while accounting for offloading overheads. The second model, referred to as power savings with wastage, aims to maximize power savings while considering power wastage associated with unsuccessful offloads. The

third model, termed fraction of completed jobs with execution time reduction, aims to maximize the fraction of jobs that reduce their execution time by offloading some or all of their work. For the first two revenue models, as a comparative baseline, we derive an upper bound on the revenue under *any* offloading and scheduling policy.

- We propose several classes of offloading policies. They differ in terms of (1) their requirements of knowledge of the system state, including adapting to the long-term offloading offered load and job types; (2) leveraging measurement-based abandonment policy, which reacts dynamically to congestion resulting from an excessive number of active users or poor wireless channel capacity relative to the job deadlines; and (3) their ability to adapt decisions regarding the choice of cut location and the fraction of jobs to admit based on changes in long term loads. We evaluate and compare these policies using representative job profiles from [36]. We do this for a range of offered job rates resulting in varying levels of congestion on the wireless network and edge server and study how close their performance is to the performance upper bound. Since some policies, such as PACCA + PA, our best policy, require prior knowledge of loads and thus possibly re-optimization when loads change, we also evaluate it under imperfect estimates of offered loads showing the approach is robust to such errors.

The chapter is organized as follows: In Section 2.2, we introduce our

basic system model. In Section 2.3, we develop an upper bound on the achievable revenue rate and explore different offloading management policies. We end the section with some simulation results. We discuss offloading management policies for heterogeneous compute job types in Section 2.4. Finally, Section 2.5 concludes the chapter.

2.2 System Model

We begin by introducing our system model for a set of users, generating homogeneous jobs (identical deadlines, cut locations, and computation-communication requirements per cut location) with limited local computation resources and a limited amount of shared wireless network and edge/cloud computation resources. We later consider heterogeneous jobs.

2.2.1 Model for load

We let $\mathcal{U}$ denote a set of users sharing a wireless access point – the set has cardinality $N = |\mathcal{U}|$. Each user u generates homogeneous jobs according to a stationary process. Users can have different channel qualities/classes. We let $\mathcal{C}$ denote the set of possible channel qualities with associated capacities. We let c_u denote the channel quality of user u and use λ_{u,c_u} to denote the arrival rate of jobs from user u. The total arrival rate of jobs from users with channel quality c is given by $\lambda_c = \sum_{u\in\mathcal{U}} \lambda_{u,c_u} \mathbf{1}(c_u = c)$. We denote the vector of total arrival rates for each channel quality as $\boldsymbol{\lambda} = (\lambda_c, c \in \mathcal{C})$. Finally, we define the total job arrival rate to the system as $\lambda = \sum_{c\in\mathcal{C}} \lambda_c$.

2.2.2 Job model

The execution of a job may involve computation on a user's device, offloading of data to the edge/cloud server, processing on the edge/cloud server, and then transmitting the result back to the device. Initially, we focus on a single job type with a fixed time budget, τ, for job execution – that does not include the time required to transmit the result back to the user device[2]. We model the job as a DAG, where the possible cut locations are denoted by a set $\mathcal{S} = \{1, 2, ..., n\}$, with n representing the last cut location. For a cut location $k \in \mathcal{S}$, we let β_k denote the cumulative device processing measured in *floating point operations (FLOPs)* including overhead related to cutting itself, d_k denotes the offload data volume (in bits), and γ_k models the cumulative edge server processing (in FLOPs). Here, $k = 1$ corresponds to processing everything on the edge server, and $k = n$ corresponds to processing everything on the user's device.

2.2.3 Model for user's device, wireless channels, and edge server resources

A user's device has an effective processing speed denoted by δ, measured in *floating point operations per second (FLOPs/sec)*. A user with channel quality c has an uplink capacity to the base station of r_c Mbps. However, the transmission rate throughout the offloading process, as explained later, may

[2]We neglect this because for a variety of applications such as video analytics, data volume associated with the result is much smaller than the uplink data transfer and the downlink capacity is typically much higher than the uplink capacity.

be reduced by congestion, e.g., by competing job offloads from other users.

We consider multiple processors with multiple cores at the edge server. The total processing capacity, ω (FLOPs/sec), is modeled as the sum of all cores' processing rates across all processors, assuming jobs can be parallelized on all processors and cores. Thus, all active jobs get an "equal" time of edge server. Note that modern computing systems would allow the parallelization of jobs across only a limited number of cores of a given processor. Hence this is a simplification.

We let $\mathcal{S}_c \subseteq \mathcal{S}$ denote the set of cut locations for a user with channel quality c that guarantee the job will meet its delay deadline, τ when one optimistically assumes there is no competition for communication or computation resources in the system. Thus k is in $\mathcal{S}_c$ if

$$\underbrace{\frac{\beta_k}{\delta}}_{\text{local processing}} + \underbrace{\frac{d_k}{r_c}}_{\text{data offloading}} + \underbrace{\frac{\gamma_k}{\omega}}_{\text{edge processing}} \leq \tau \tag{2.1}$$

where the left-hand side is the best possible end-to-end time to complete the offload when a job is cut at location k.

2.2.4 Sharing base station uplink resources

At any time, t, multiple users may be offloading data. $\mathcal{U}(t)$ represents the set of active users, and $N(t) = |\mathcal{U}(t)|$ is its cardinality. We shall assume that all users in $\mathcal{U}(t)$ share the BS's uplink resources in a Proportional Fair manner, with each ongoing offloading over channel c served at rate $r_c \frac{1}{N(t)}$.

2.2.5 Model for computation on the device and data offloading

When a job with local computational requirements is generated on a user's device, it undergoes computation based on a non-preemptive priority scheme that prioritizes jobs by their generation time. As a result, a job may be queued before its execution. After it leaves the queue, it is processed if there is enough time to execute it; otherwise, it is dropped.

Once the local part of the execution is complete, a job with data to offload is served on a first-come, first-served basis, so there may be additional waiting before offloading to the edge server begins.

2.2.6 Stationary offloading policies

We consider a set Π of stationary offloading policies. A policy may consist of *any* combination of job admission control/cutting /offloading methods, wireless channel scheduling, and edge server resource sharing. For a given offered load $\boldsymbol{\lambda}$ and policy $\pi \in \Pi$, we define $\mathbf{q}(\boldsymbol{\lambda}, \pi) = (q_{c,k}(\boldsymbol{\lambda}, \pi) : c \in \mathcal{C}, k \in \mathcal{S})$, where $q_{c,k}(\boldsymbol{\lambda}, \pi)$ denotes the long-term *fraction* of jobs that belong to users with channel quality c, are cut at location k, and complete. Naturally, it must be the case that the sum of these fractions is less than or equal to one, meaning $\sum_{k \in \mathcal{S}} q_{c,k}(\boldsymbol{\lambda}, \pi) \leq 1$ for all $c \in \mathcal{C}$ (in case not all jobs complete on time). We define $\mathcal{F}(\boldsymbol{\lambda}) = \{\mathbf{q}(\boldsymbol{\lambda}, \pi) \mid \pi \in \Pi\}$, as the set of fractions that are feasible under some policy. This set is convex since if $\pi_1, \pi_2 \in \Pi$, then by alternating between policies over long periods, one can achieve any convex combination of their associated performance.

2.2.7 Reward model and revenue metric

We introduce three reward models to guide the design and evaluation of offloading policies. We use α_k to represent the reward associated with the timely completion of a job cut at location k.

Net timely offloaded workload. The first reward model captures the total amount of work offloaded to the edge server for jobs that complete on time. It indirectly captures the freeing up of users' computation resources. The reward for offloading a job at cut location k is modeled as

$$\alpha_k = \gamma_k - g \cdot d_k. \tag{2.2}$$

Here γ_k denotes computation work offloaded to the edge and $g \cdot d_k$ the overhead of doing so, where d_k represents the volume of data offloaded, and g is a factor that "converts" bits to FLOPs.

The *net timely offloaded workload* measured in FLOPs/sec for a given offered load $\boldsymbol{\lambda}$ under policy π, is defined as

$$O(\mathbf{q}(\boldsymbol{\lambda}, \pi), \boldsymbol{\lambda}) = R_{\text{ow}}(\mathbf{q}(\boldsymbol{\lambda}, \pi), \boldsymbol{\lambda}) - L_{\text{ow}}(\boldsymbol{\lambda}, \pi) \tag{2.3}$$

where

$$R_{\text{ow}}(\mathbf{q}(\boldsymbol{\lambda}, \pi), \boldsymbol{\lambda}) = \sum_{c \in \mathcal{C}} \lambda_c \sum_{k \in \mathcal{S}} \alpha_k q_{c,k}(\boldsymbol{\lambda}, \pi) \tag{2.4}$$

denotes the rate at which net work is offloaded, and $L_{\text{ow}}(\boldsymbol{\lambda}, \pi)$ is the rate of computational work on users' devices associated with jobs that do not complete

on time and with jobs that a user attempts to offload but ends up completing locally[3].

Power savings with wastage. The second reward model quantifies the energy savings on a user device resulting from offloading. The energy expended by a device when offloading a job at cut k is modelled as $a \cdot \beta_k + b \cdot d_k$ joules, where a and b represent the energy expended per FLOP for local computation and per bit for data offloading, respectively. The energy savings from offloading at cut k vs. not offloading at all, i.e., cut at n (the last cut location), is given by

$$\alpha_k = a \cdot (\beta_n - \beta_k) + b \cdot (d_n - d_k) \tag{2.5}$$

in joules. This captures the energy saved from decreased local computation while considering the energy overhead of data offloading.

We define the net *power savings with wastage* measured in Watts for a given load $\boldsymbol{\lambda}$ under policy π, as

$$P(\mathbf{q}(\boldsymbol{\lambda}, \pi), \boldsymbol{\lambda}) = R_{\text{ps}}(\mathbf{q}(\boldsymbol{\lambda}, \pi), \boldsymbol{\lambda}) - L_{\text{ps}}(\boldsymbol{\lambda}, \pi) \tag{2.6}$$

where $R_{\text{ps}}(\cdot)$ is defined in the same way as $R_{\text{ow}}(\cdot)$ but with the energy savings reward α_k defined above for each timely job completion. $L_{\text{ps}}(\boldsymbol{\lambda}, \pi)$ represents the power expended at devices associated with jobs that miss their deadlines and with jobs that a user attempts to offload but ends up completing locally.

[3]Note this occurs when a user attempts to offload a job but due to congestion on wireless channels and/or edge server abandons and reverts to local execution.

Fraction of Completed jobs with an execution time Reduction of at least 1/s (FCR(s)). The third reward model captures the fraction of jobs that experience reduced execution times due to offloading some or all of their sub-tasks. A job takes $\frac{\beta_n}{\delta}$ seconds for local execution, where n represents the last cut location. We define $\tau_s = \frac{1}{s}\frac{\beta_n}{\delta}$ as the reduced execution time, with 1/s ($0 < 1/s \leq 1$) signifying the the fraction of time it takes to execute the job when some or all of its sub-tasks are offloaded compared to executing it locally. Given an offered load $\boldsymbol{\lambda}$, reduced execution time τ_s, and offloading policy π, we define $\mathbf{q}(\boldsymbol{\lambda}, \pi, \tau_s) = (q_{c,k}(\boldsymbol{\lambda}, \pi, \tau_s) : c \in \mathcal{C}, k \in \mathcal{S} \setminus \{n\})$, where $q_{c,k}(\boldsymbol{\lambda}, \pi, \tau_s)$ denotes the long-term *fraction* of jobs associated with users with channel quality c that are cut at location k, and complete within τ_s.

We define the overall fraction of jobs that achieve an execution time reduction of at least 1/s for a given load $\boldsymbol{\lambda}$ under policy π, as

$$S(\mathbf{q}(\boldsymbol{\lambda}, \pi, \tau_s), \boldsymbol{\lambda}, \tau_s) = \sum_{c \in \mathcal{C}} \sum_{k \in \mathcal{S} \setminus \{n\}} q_{c,k}(\boldsymbol{\lambda}, \pi, \tau_s). \tag{2.7}$$

For this metric, we assume that every job would be executed and completed locally[4] within $\frac{\beta_n}{\delta}$, the time required for local execution, although some users may concurrently offload at a pre-selected cut location to possibly reduce their execution time. Note that this metric optimistically ignores the slowdown caused by concurrently offloading data while executing the job locally.

[4]Thus, we ignore any cost (as done in previous revenue metrics) corresponding to jobs that miss the τ_s execution deadline.

2.3 Homogeneous Jobs and Offloading Policies

In this section, we propose and evaluate several offloading policies for users with heterogeneous channel qualities but a homogeneous job type. In the next section, we extend the analysis to the case with heterogeneous job types.

2.3.1 Upper bound

We begin by developing a simple upper bound for the net timely offloaded workload or power savings with wastage achievable by any stationary offloading policy. Let $\mathbf{q} = (q_{c,k} : c \in \mathcal{C}, k \in \mathcal{S})$, where $q_{c,k}$ denotes the fraction of jobs that belong to users with channel quality c, are cut at location k, and complete. We define the set of all possible vectors $\mathbf{q}$ as

$$\begin{aligned} \mathcal{Q} = \{\mathbf{q} \mid \mathbf{q} \geq \mathbf{0}, \sum_{k \in \mathcal{S}} q_{c,k} \leq 1 \ \forall \ c \in \mathcal{C} \\ \text{and } q_{c,k} = 0, \ c \in \mathcal{C}, \ k \in \mathcal{S} \setminus \mathcal{S}_c\} \end{aligned} \tag{2.8}$$

where in some settings, a fraction of jobs may not complete, hence they need not sum up to 1, and fractions for infeasible cut locations are zero. Given $\mathbf{q}$, we define the channel and edge server utilization as

$$\rho_{\text{ch}}(\mathbf{q}) = \sum_{c \in \mathcal{C}} \lambda_c \sum_{k \in \mathcal{S}} q_{c,k} \frac{d_k}{r_c} \mathbf{1}1(k \in \mathcal{S}_c), \tag{2.9}$$

$$\rho_{\text{ed}}(\mathbf{q}) = \sum_{c \in \mathcal{C}} \lambda_c \sum_{k \in \mathcal{S}} q_{c,k} \frac{\gamma_k}{\omega} \mathbf{1}1(k \in \mathcal{S}_c), \tag{2.10}$$

respectively. Recall that we defined $\mathcal{F}(\boldsymbol{\lambda})$ to be the set of *feasible* long-term fractions of successful job completions under a set of stationary offloading

policies when the system load is $\boldsymbol{\lambda}$. We define the set of all *possible* successful long term fractions

$$\overline{\mathcal{F}}(\boldsymbol{\lambda}) = \{\mathbf{q} \mid \mathbf{q} \in \mathcal{Q},\ \rho_{\text{ch}}(\mathbf{q}) \leq 1 \text{ and } \rho_{\text{ed}}(\mathbf{q}) \leq 1\} \tag{2.11}$$

as a natural outer bound for $\mathcal{F}(\boldsymbol{\lambda})$ which leads to the following simple performance bounds.

Theorem 1. Given an offered load $\boldsymbol{\lambda}$ we have that $\mathcal{F}(\boldsymbol{\lambda}) \subseteq \overline{\mathcal{F}}(\boldsymbol{\lambda})$. Then the maximum net timely offloaded workload achievable by any stationary offloading policy is defined as

$$O^*(\boldsymbol{\lambda}) := \max_{\mathbf{q} \in \mathcal{F}(\boldsymbol{\lambda})} O(\mathbf{q}, \boldsymbol{\lambda}) \leq \max_{\mathbf{q} \in \overline{\mathcal{F}}(\boldsymbol{\lambda})} R_{\text{ow}}(\mathbf{q}, \boldsymbol{\lambda}) \tag{2.12}$$

Similarly, the maximum power savings with wastage achievable by any stationary offloading policy is defined as

$$P^*(\boldsymbol{\lambda}) := \max_{\mathbf{q} \in \mathcal{F}(\boldsymbol{\lambda})} P(\mathbf{q}, \boldsymbol{\lambda}) \leq \max_{\mathbf{q} \in \overline{\mathcal{F}}(\boldsymbol{\lambda})} R_{\text{ps}}(\mathbf{q}, \boldsymbol{\lambda}) \tag{2.13}$$

Proof. We first argue that $\mathcal{F}(\boldsymbol{\lambda}) \subseteq \overline{\mathcal{F}}(\boldsymbol{\lambda})$. Indeed suppose $\mathbf{q} \in \mathcal{F}(\boldsymbol{\lambda})$. Recall that $q_{c,k}$ represents the fraction of incoming jobs that belong to users with channel quality c, are cut at location k, and complete. Since each job is cut at only one location, these fractions always sum to less than or equal to 1 over all cut locations, thus q is clearly in $\mathcal{Q}$ – but suppose the load on the channel or edge server is greater than 1 – given these fraction of jobs, i.e., $\rho_{\text{ch}}(\mathbf{q}) > 1$ or $\rho_{\text{ed}}(\mathbf{q}) > 1$. If this is true, then not all jobs will complete on time, contradicting our earlier statement. Thus, the channel and edge

server load must be less than or equal to 1, i.e., $\rho_{\text{ch}}(\mathbf{q}) \leq 1$ and $\rho_{\text{ed}}(\mathbf{q}) \leq 1$ if $\mathbf{q} \in \mathcal{F}(\boldsymbol{\lambda})$, which implies $\mathbf{q} \in \overline{\mathcal{F}}(\boldsymbol{\lambda})$. Thus $\mathcal{F}(\boldsymbol{\lambda}) \subseteq \overline{\mathcal{F}}(\boldsymbol{\lambda})$. This then implies $\max_{\mathbf{q}\in\mathcal{F}(\boldsymbol{\lambda})} R_{\text{ow}}(\mathbf{q}, \boldsymbol{\lambda}) \leq \max_{\mathbf{q}\in\overline{\mathcal{F}}(\boldsymbol{\lambda})} R_{\text{ow}}(\mathbf{q}, \boldsymbol{\lambda})$ which results in the Equation 2.12, since

$L_{\text{ow}}(\boldsymbol{\lambda}, \pi) \geq 0$. Similarly under the energy savings reward model and recognizing that $L_{\text{ps}}(\boldsymbol{\lambda}, \pi) \geq 0$, we have Equation 2.13. □

Remark 1. We let $\mathbf{q}^*_{\text{ow}}(\boldsymbol{\lambda}) = \text{argmax}_{\mathbf{q}\in\overline{\mathcal{F}}(\boldsymbol{\lambda})} R_{\text{ow}}(\mathbf{q}, \boldsymbol{\lambda})$ and

$\mathbf{q}^*_{\text{ps}}(\boldsymbol{\lambda}) = \text{argmax}_{\mathbf{q}\in\overline{\mathcal{F}}(\boldsymbol{\lambda})} R_{\text{ps}}(\mathbf{q}, \boldsymbol{\lambda})$ denote the vector that maximizes the bounds for the two reward models. These indicate the fraction of load to admit across sets of channel qualities and cut locations to maximize revenue in the absence of resource contention during job offloading.

2.3.2 Offloading policies

Naive Greedy (NG). The NG offloading policy optimistically assigns the cut location, k_u, that yields the highest reward among all feasible cut locations given deadline τ, to all jobs generated by user u in $\mathcal{U}$ with channel quality c_u as follows:

$$k_u = \underset{k\in\mathcal{S}_{c_u}}{\text{argmax}}\, \alpha_k \tag{2.14}$$

where α_k either reflects net offloaded workload or energy savings, see Section 2.2.7. The policy then greedily tries to offload data and process on the edge server until success or time budget, τ, expires.

Predictive Abandonment (PA). PA is a real-time user-based of-

floading policy that adapts to channel and edge/cloud server congestion. Similar to the NG policy, a user u implementing PA selects the cut location k_u with the highest reward, see Equation 2.14. However, unlike NG, during the offloading process, a user implementing PA estimates the feasibility of meeting a job's completion deadline given current channel and/or edge server congestion. If the deadline is unlikely to be met, PA saves resources by abandoning the job's offload, thus increasing the likelihood of completing other jobs on time. Furthermore, a job whose offload is abandoned can still attempt to complete its residual processing on the user device if time permits. Thus, PA can be viewed as performing a type of state-dependent self-admission control or abandonment policy.

Under PA, a user u determines whether its ith job is likely to complete on time by considering/predicting the following factors: queuing time, local processing time, data offloading time, and edge processing time. The queuing time for a job i at time t, denoted $W_{u,i}(t)$, corresponds to the time the job has been waiting in the user's queue. The local processing time is calculated as the number of operations before the cut location k_u divided by the device's execution speed, i.e., $\frac{\beta_{k_u}}{\delta}$ secs. Since a user's compute speed is fixed, this value is the same for all t. Also, recall that all jobs from a user u are cut at the same location. The data offloading time at time t is calculated as the sum of time already spent offloading the job (if any) and the time required to offload any remaining data. The latter can be estimated by dividing the data yet to be offloaded at time t, $s_{u,i}(t)$, by an estimate for the future average transmission

rate $h_{u,i}(t)$. Suppose user u initiates data offloading of job i at time $t_{u,i}$ over channel quality c_u. At any instance t' (for t' greater than or equal to $t_{u,i}$), the transmission rate under our model is given by $\frac{r_{c_u}}{N(t')}$, where recall r_{c_u} is the uplink channel capacity and $N(t')$ is the number of active users. We estimate the future average transmission rate based on the average throughput experienced by the user u since it began offloading job i, i.e.,

$$h_{u,i}(t) = \frac{1}{e_{u,i}(t)} \sum_{t'=t_{u,i}}^{e_{u,i}(t)+t_{u,i}} \frac{r_{c_u}}{N(t')} \tag{2.15}$$

where $e_{u,i}(t)$ is the time elapsed since offloading began. The last factor in the equation for job latency is the edge processing time. We assume that the edge server provides users with or users themselves maintain estimates of the job's processing time per cut location, $\overline{m}_{k_u}$. These estimates are periodically updated.

Putting together the elapsed time and estimated future transmission/processing latencies, the estimated total latency for user u's job i is given by

$$\underbrace{W_{u,i}(t)}_{\text{queuing}} + \underbrace{\frac{\beta_{k_u}}{\delta}}_{\text{local processing}} + \underbrace{e_{u,i}(t) + \frac{s_{u,i}(t)}{h_{u,i}(t)}}_{\text{data offloading}} + \underbrace{\overline{m}_{k_u}}_{\text{edge processing}} . \tag{2.16}$$

Under PA, a user may abandon offloading if it determines that its estimated total latency for job i is greater than its time budget, τ. Once abandoned, any remaining computation, γ_{k_u}, for job i can be completed on the user's device if there is enough time.

This prediction method is crude but roughly captures the impact of the user's channel capacity r_{c_u}, channel's uplink congestion $N(t)$, and congestion at the edge server using the actual estimate of processing latencies, $\overline{m}_{k_u}$. Note that these estimates will reflect changes resulting from the PA policy itself since PA impacts the load on the channel and edge server. For more accurate predictions, one can consider additional factors such as number of active users during each user's offload and/or remaining service requirements of currently offloading jobs. See, [63] for such a discussion in a processor-sharing scenario without abandonment.

PA effectively performs a sort of "real-time" admission control by abandoning jobs unlikely to meet their deadlines due to channel and/or edge server congestion. In congested scenarios, PA "prefers" users with better channels, since they are more likely to complete their offloads on time. Finally, note PA, like NG, is decentralized and does not require knowledge of the overall offered job rate, $\boldsymbol{\lambda}$. Next, we explore a policy that considers the overall rate of offered jobs for coordination.

Probabilistic Admission Control and Cut Assignment (PACCA). Under PACCA, we pre-determine $p_{c,k}$, the fraction of jobs that belong to users with channel quality c that should be offloaded at each cut location k to maximize revenue given the rate of incoming jobs. We let $\mathbf{p} = (p_{c,k} : c \in \mathcal{C}, k \in \mathcal{S})$, and require $\sum_{k \in \mathcal{S}} p_{c,k} \leq 1$ for all $c \in \mathcal{C}$. We denote the associated policy as PACCA ($\mathbf{p}$). If a job is not admitted for offloading under this policy, it will attempt to execute locally. Determining

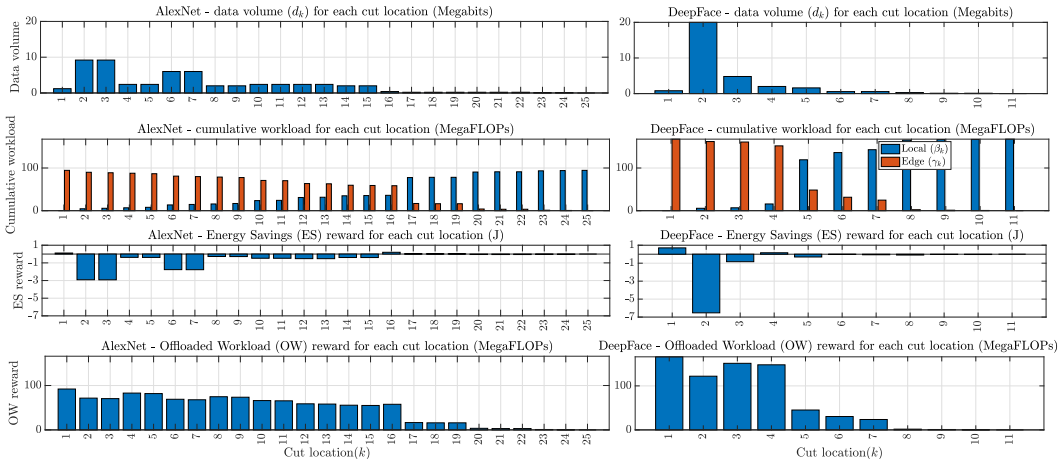


Figure 2.2: Simulation parameters associated with AlexNet and DeepFace job types.

$\mathbf{p}$ is complex, as it involves multiple factors, such as contending offloads from users with different channel qualities, channel and server congestion, delay constraints, and revenue rate optimization. We propose to choose $\mathbf{p}$ based on the upper bounds from Theorem 1, either $\mathbf{q}^*_{\text{ow}}(\boldsymbol{\lambda})$ to maximize net timely offloaded workload or $\mathbf{q}^*_{\text{ps}}(\boldsymbol{\lambda})$ to maximize power savings with wastage. In later sections, for brevity we use $\mathbf{q}^*$ to refer to either $\mathbf{q}^*_{\text{ow}}(\boldsymbol{\lambda})$ or $\mathbf{q}^*_{\text{ps}}(\boldsymbol{\lambda})$. Given that there is no resource contention underlying Theorem 1 (see Remark 1), these probabilities reflect optimistic admission control and cut assignment for a given $\boldsymbol{\lambda}$. Nevertheless, they still reflect reasonable overall system tradeoffs.

Once a job is admitted for offloading, the user can either attempt to offload the job greedily at the assigned cut location or perform PA, only proceeding with the offload if it determines the job's deadline can be met. We refer to the former policy as PACCA ($\mathbf{q}^*$) + Greedy and the latter as PACCA

$(\mathbf{q}^*)$ + PA.

2.3.3 Simulation results

In this subsection, we evaluate the performance of our proposed offloading policies via discrete-time simulations in terms of: (i) net timely offloaded workload; (ii) power savings with wastage (explained in Section 2.2.7). We also plot the fraction of jobs completed. The simulations are conducted in MATLAB R2023a.

Settings. We consider a system with $N = 20$ users[5], where each user generates an equal rate of homogeneous jobs per second, according to a Poisson distribution[6], with intensity $\lambda/20$. This results in an aggregate job generation/arrival rate of λ per second. The system includes two channel qualities, half of the users (i.e., 10) have one channel quality, half the other[7]. A user's channel quality/capacity does not change, but the transmission rate at any instance depends on both the channel capacity and the number of competing users because of the Proportional Fair sharing of uplink resources. Table 2.1 summarizes the simulation parameters. We present results for the homogeneous scenario based on the job characteristics of AlexNet, a state-of-the-art Convolutional Neural Network for image classification. In the next section, we will also use DeepFace, which is used for face recognition. Fig. 2.2

[5]We chose this to represent typical traffic at a 5G BS/AP in a dense urban environment deployment scenario, we can increase/decrease the number as needed.

[6]We also ran the simulations for other arrival processes but due to space limitations only include the results for the Poisson arrival process.

[7]We consider this for simplicity; our work applies to any number of channel qualities.

displays the job characteristics of AlexNet on the left and DeepFace on the right. The four bar graphs (from top to bottom) show the volume of data that gets offloaded, the amount of local vs. edge processing, the energy saved, and the amount of work that gets offloaded per cut location. The worst delay a job may experience is calculated as $\tau_{\max} = \max_k \left(\frac{\beta_k}{\delta} + \frac{d_k}{\min_{c \in \mathcal{C}} r_c} + \frac{\gamma_k}{\omega} \right)$. However, this may not be the absolute worst case as it only considers the worst channel quality and not congestion. We evaluated our policies under both strict $\tau = 0.4\tau_{\max}$ and relaxed $\tau = 0.8\tau_{\max}$ delay deadlines. Results are averaged over 20 Monte Carlo simulations, each performed over 5e5 time slots. Here a time slot is 100 µs long.

Table 2.1: Simulation parameters

Parameter (units)	Value	Parameter (units)	Value
r_c (Mbps)	$(20, 40)$	g (FLOP/bit)	2
δ (FLOPs/sec)	$1125 * 10^6$	a (J/FLOP)	$6 * 10^{-9}$
ω (FLOPs/sec)	$11360 * 10^6$	b (J/bit)	$4 * 10^{-7}$

Results discussion. Our first set of results, presented in Figures 2.3a and 2.3b, include the net timely offloaded workload and fraction of total jobs completed for our policies per total offered job rate, respectively, under a strict job deadline. In Fig. 2.3a, we show the net computational workload offloaded, which depends on the fraction of total jobs completed and the reward per completed job. Therefore, a policy can perform equally well in two cases: (i) completing numerous jobs with a low reward or (ii) completing fewer jobs

with a high reward. Thus, we also present the fraction of completed jobs in Fig. 2.3b for the same simulation setting for all policies.

We observe that as the offered job rate λ increases, the NG policy experiences throughput collapse, see Fig. 2.3a. By contrast, policies like PA, which implement congestion-dependent offload abandonment, and PACCA, which determines admission control and cut assignments based on prior knowledge of incoming jobs per user and channel quality, perform well under heavy job loads. However, PA does not perform as well as PACCA, highlighting the benefit reserving channel and edge resources for jobs with good channels and/or high reward cut locations. The benefit of combining these policies, PACCA + PA becomes more evident at high-load regimes where admission control and congestion management is crucial.

In Fig. 2.3b, we show the fraction of completed jobs under different offloading policies as job arrival rate λ increases. With PA, more than 90% of jobs are completed in the load regimes considered. Indeed, for all the results reported hereafter, we only considered load regimes where PA has a high completion fraction (at least 90%) and where the channel is the bottleneck. Interestingly, the fraction of jobs that complete under PACCA + Greedy is non-monotonic with increasing load. This is because initially (from 0 to 23 jobs/sec), PACCA selects the highest reward cut location for every job. However, since channel capacity is fixed, an increase in the number of jobs means a decrease in completions. Hence, the downward curve. Then, at 23 jobs/sec, PACCA determines it will earn more revenue by adjusting the distribution of

cut locations, so that jobs have less data to offload. This results in less reward per job, but more completed jobs. PACCA makes this adjustment every time the rate of incoming jobs increases beyond 23 jobs/sec, thus the upward curve. We observe similar non-monotonic behavior for PACCA + PA though it is barely perceptible in the figure.

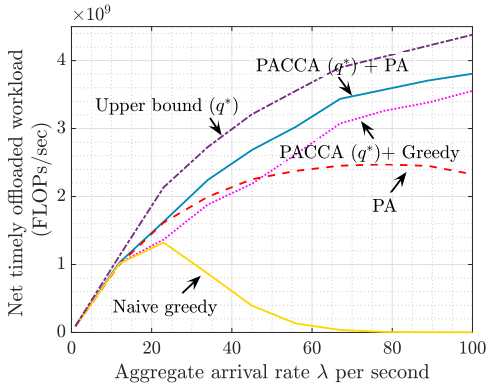
In Fig. 2.3a, we saw the advantage of combining PACCA with PA under a strict delay deadline. We observe similar benefits under a relaxed delay deadline, see Fig. 2.4a; however, the performance gap between PACCA + PA and PACCA + Greedy decreases. This is because, with a relaxed delay deadline, a user has a higher chance of completing an offload, even with network congestion. Thus, network congestion is detrimental only under strict delay deadlines necessitating a congestion-aware policy like PA. We see similar trends with power savings with wastage in Figures 2.5a and 2.6a. Additionally, as we relax the completion deadline, our best-performing policy (PACCA + PA) approaches the upper bound for both performance metrics.

We now compare different offloading policies based on the Fraction of Completed jobs with an execution time Reduction of at least 1/s or FCR(s), as introduced in Eq. 2.7. In the simulation results presented in Figures 2.7 and 2.8, all users have the same channel capacity, specifically, $r_c = 40$ Mbps. For the PA/NG policy, we predetermine the optimal cut location as the cut that results in minimal delay in the best-case scenario (no congestion anywhere), i.e., $k_u = \text{argmax}_{k \in \mathcal{S} \setminus \{n\}} \left(\frac{\beta_k}{\delta} + \frac{d_k}{r_c} + \frac{\gamma_k}{\omega} \right)$. We use the same cut location for PACCA, although we additionally optimize the fraction of jobs admitted for

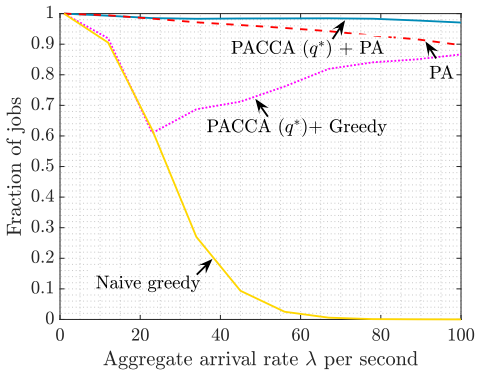
offloading such that neither the channel nor edge compute utilization exceeds 1. In Fig. 2.7a, we present the results for different values of $1/s$, the ratio of reduced execution time τ_s to local execution time $\frac{\beta_n}{\delta}$. These results are for an aggregate arrival rate of $\lambda = 30$. As expected, for small values of of $1/s$, few jobs are able to reduce their execution times that much even with offloading. However, for large values of $1/s$, the fraction of jobs that experience a small reduction in execution time increases significantly. In the plot shown, the performance of PACCA + PA and PA coincides, and these policies outperform their greedy counterparts. In addition, we also exhibit the fraction of abandoned jobs under the abandonment based policies in Fig. 2.7b, where we observe an overlap in the fraction of total jobs abandoned.

We see similar results for an aggregate arrival rate, $\lambda = 50$ in Fig. 2.8a. In addition to a decrease in the fraction of jobs experiencing reduced execution times as compared to results for $\lambda = 30$, we now see a performance gap between policies that perform admission control and cut assignment versus those that do not. Given that we do not penalize the final performance of policies based on the fraction of abandoned jobs, we see PA outperforming PACCA + PA even though a significant fraction of jobs are abandoned under PA, as shown in Fig. 2.8b.

Robustness study. We demonstrate the robustness of PACCA + PA to imperfect knowledge of offered job load in Figures 2.9a and 2.9b, for net timely offloaded workload under a strict and relaxed delay deadline, respectively. In these simulations, we added errors to the estimates of job arrival

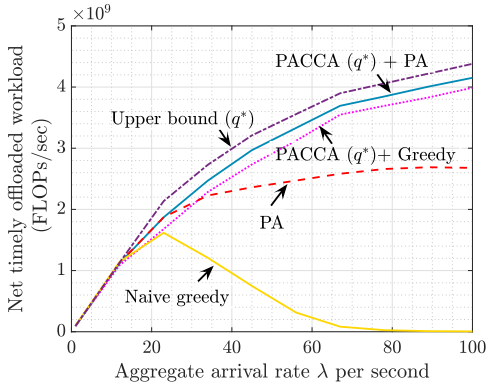


(a) Net timely offloaded workload (FLOPs/sec).

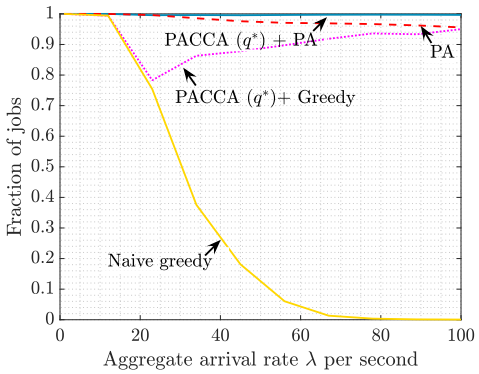


(b) Fraction of total jobs completed.

Figure 2.3: Comparing the net timely offloaded work and fraction of total jobs that complete for different policies when the job's delay deadline is strict, i.e., $\tau = 0.4\tau_{\max}$.

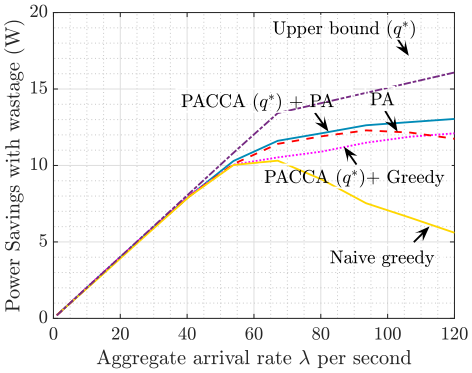


(a) Net timely offloaded workload (FLOPs/sec).

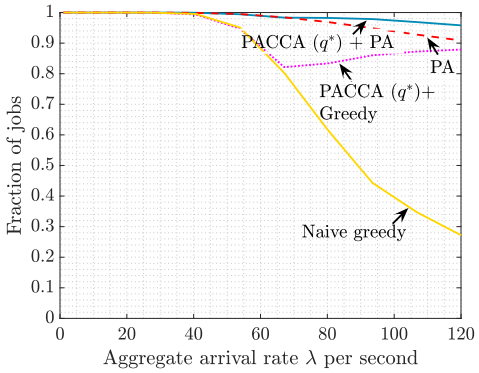


(b) Fraction of total jobs completed.

Figure 2.4: Comparing the net timely offloaded work and fraction of total jobs that complete for different policies when the job's delay deadline is relaxed, i.e., $\tau = 0.8\tau_{\max}$.

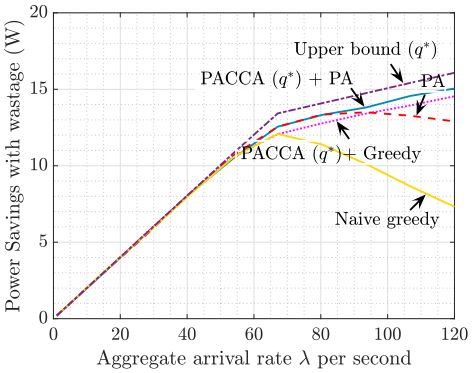


(a) Power savings with wastage (W).

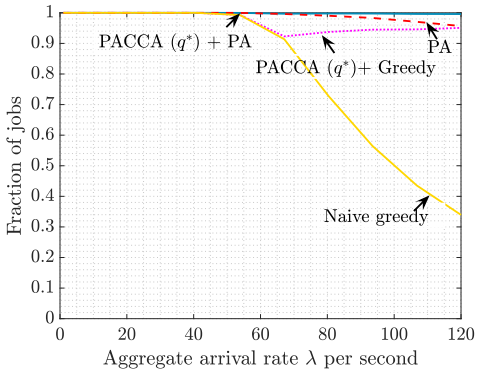


(b) Fraction of total jobs completed.

Figure 2.5: Comparing the power savings with wastage and fraction of total jobs that complete for different policies when the job's delay deadline is strict, i.e., $\tau = 0.4\tau_{\max}$.

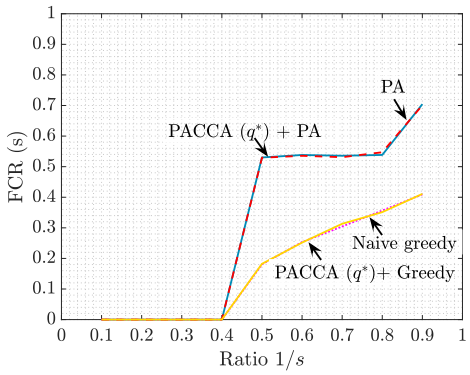


(a) Power savings with wastage (W).

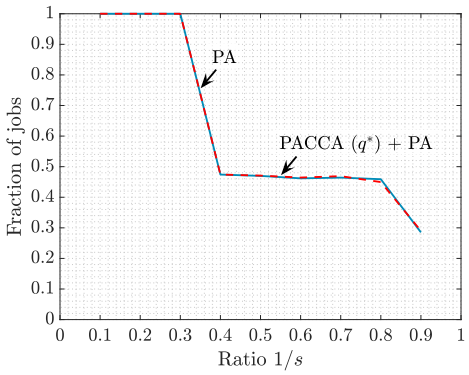


(b) Fraction of total jobs completed.

Figure 2.6: Comparing the power savings with wastage and fraction of total jobs that complete for different policies when the job's delay deadline is relaxed, i.e., $\tau = 0.8\tau_{\max}$.

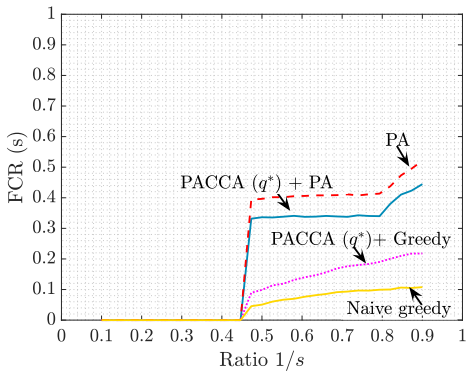


(a) Fraction of total jobs completed within τ_s.

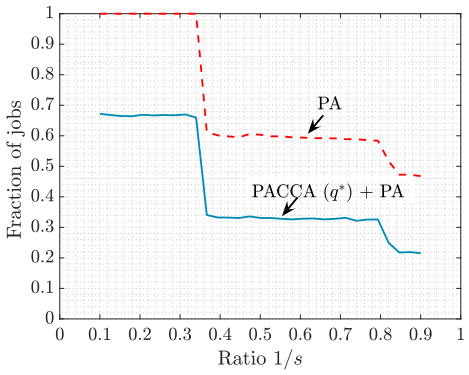


(b) Fraction of total jobs abandoned.

Figure 2.7: Comparing the fraction of jobs that experience a reduced execution time of at least $\tau_s = \frac{1}{s}\frac{\beta_n}{\delta}$ under an offloading policy for $\lambda = 30$.



(a) Fraction of total jobs completed within τ_s.



(b) Fraction of total jobs abandoned.

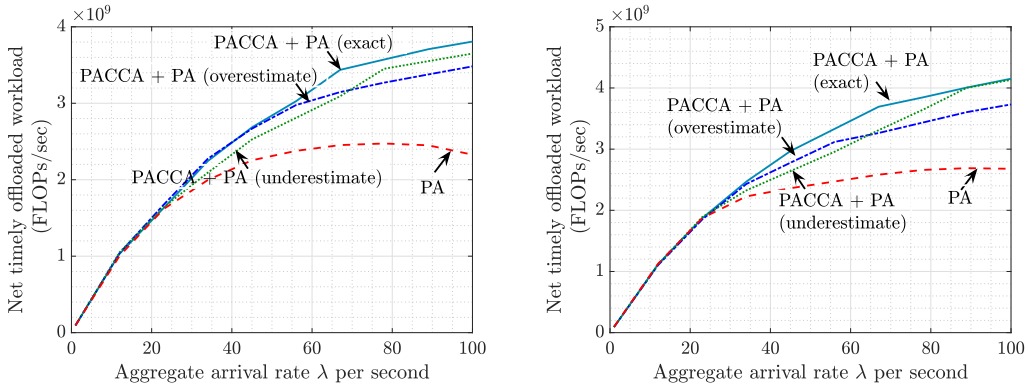
Figure 2.8: Comparing the fraction of jobs that experience a reduced execution time of at least $\tau_s = \frac{1}{s}\frac{\beta_n}{\delta}$ under an offloading policy for $\lambda = 50$.

rates per user, which affect the aggregate arrival rate per channel quality. We optimize PACCA + PA for the corresponding load and compare three scenarios: (i) PACCA + PA (exact), where we provide PACCA with the exact load, i.e., λ ; (ii) PACCA + PA (overestimate), where we provide PACCA with an overestimate of load, i.e., $\lambda \cdot (1+x\%)$, leading to less offloading compared to (i); and (iii) PACCA + PA (underestimate), where we provide PACCA with an underestimate of load, i.e., $\lambda \cdot (1-x\%)$, resulting in more offloading compared to (i). We show PA as a baseline. The results show that for an estimation error of 25% (i.e., $x = 25$), both PACCA + PA (overestimate) and PACCA + PA (underestimate) are within 10% of PACCA + PA (exact). Additionally, we observe that PACCA + PA (underestimate) performs at least as well as PA, as huge underestimation errors result in admitting every job at the highest reward cut location, effectively imitating PA. We observe similar trends with power savings with wastage in Figures 2.10a and 2.10b.

2.4 Heterogeneous Jobs

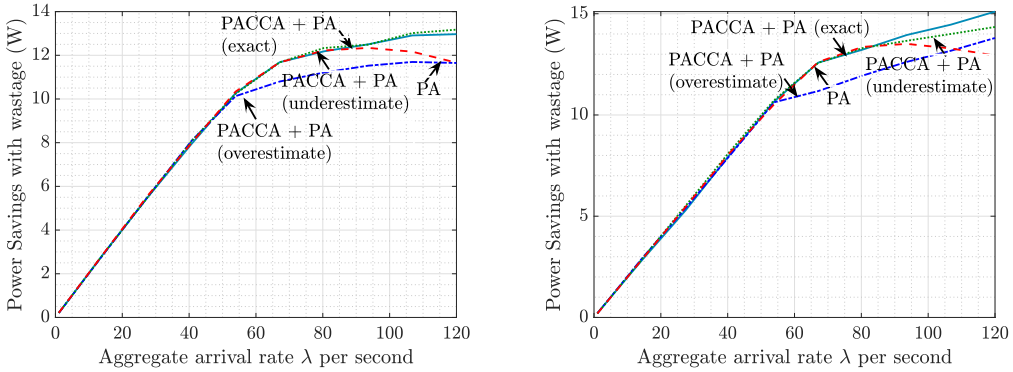
In this section, we investigate the performance of our policies in networks with heterogeneous jobs, where jobs no longer have identical deadlines, cut locations, and requirements per cut location (such as computation and data to offload).

We have a set of users generating different types of jobs, denoted by $\mathcal{J} = \{1, 2, ..., J\}$. For each job type j in $\mathcal{J}$, we let $\mathcal{U}^j$ be the set of users generating such jobs, and $\lambda^{j,u}$ denotes the arrival rate of job type j generated by



(a) Strict delay deadline, $\tau = 0.4\tau_{\max}$. (b) Relaxed delay deadline, $\tau = 0.8\tau_{\max}$.

Figure 2.9: Evaluating robustness of PACCA + PA for a 25% deviation from exact load knowledge when the job's delay deadline is strict vs. relaxed for net timely offloaded workload.



(a) Strict delay deadline, $\tau = 0.4\tau_{\max}$. (b) Relaxed delay deadline, $\tau = 0.8\tau_{\max}$.

Figure 2.10: Evaluating robustness of PACCA + PA for a 25% deviation from exact load knowledge when the job's delay deadline is strict vs. relaxed for power savings with wastage

.

user u in $\mathcal{U}^j$. The total arrival rate of job type j is denoted as $\lambda^j = \sum_{u \in \mathcal{U}^j} \lambda^{j,u}$. We define the arrival rate of job type j over channel quality c in $\mathcal{C}$ from all users as $\lambda_c^j = \sum_{u \in \mathcal{U}^j} \lambda^{j,u} \mathbf{1}(c_u = c)$, and let $\boldsymbol{\lambda}^j = (\lambda_c^j, c \in \mathcal{C})$. The arrival rate of each job type is captured by $\boldsymbol{\Lambda} = (\boldsymbol{\lambda}^1, ..., \boldsymbol{\lambda}^J)$. We use τ^j to represent the delay constraint, and $\mathcal{S}^j$ to capture the set of possible cut locations for type j jobs. As before, $\mathcal{S}_c^j$ is a subset of $\mathcal{S}^j$ that only includes cut locations that are feasible for a given user's channel quality c under time budget τ^j, i.e., a location k is in $\mathcal{S}_c^j$ if

$$\underbrace{\frac{\beta_k^j}{\delta}}_{\text{local processing}} + \underbrace{\frac{d_k^j}{r_c}}_{\text{data offloading}} + \underbrace{\frac{\gamma_k^j}{\omega}}_{\text{edge processing}} \leq \tau^j \tag{2.17}$$

where β_k^j represents the computational burden (in FLOPs) on the user (including overhead for cutting), d_k^j denotes the volume (in bits) of data transfer to the edge server, and γ_k^j captures the computational burden (in FLOPs) on the edge server for a type j job cut at location k.

For an offered load, $\boldsymbol{\Lambda}$, under offloading policy $\pi \in \Pi$, we define $\mathbf{Q}(\boldsymbol{\Lambda}, \pi) = (\mathbf{q^1}(\boldsymbol{\Lambda}, \pi), ..., \mathbf{q^J}(\boldsymbol{\Lambda}, \pi))$, where $\mathbf{q^j}(\boldsymbol{\Lambda}, \pi) = (q_{c,k}^j(\boldsymbol{\Lambda}, \pi) : c \in \mathcal{C}, k \in \mathcal{S}^j)$, and $q_{c,k}^j(\boldsymbol{\Lambda}, \pi)$ is the long-term fraction of type j jobs that belong to users with channel quality c, are cut at location k, and complete. These fractions must sum to less than or equal to 1 (some jobs may not complete), i.e., $\sum_{k \in \mathcal{S}^j} q_{c,k}^j(\boldsymbol{\Lambda}, \pi) \leq 1$ for all $j \in \mathcal{J},\ c \in \mathcal{C}$. We define $\mathcal{T}(\boldsymbol{\Lambda}) = \mathbf{Q}(\boldsymbol{\Lambda}, \pi)$ for $\pi \in \Pi$ as the set of feasible long-term fractions, which is convex through time sharing argument presented in the homogeneous case.

Just as in the homogeneous scenario, we have two revenue metrics: weighted net timely offloaded workload revenue and weighted power savings with wastage revenue. We let α_k^j denote the reward generated from a successful job completion when a type j job on a user is cut and offloaded at cut location k.

Weighted net timely offloaded workload revenue. We define it as a weighted sum of net timely offloaded workload for each job type, where w^j is the weight [8] for job type j and $\sum_{j\in\mathcal{J}} w^j = 1$. For a given offered load $\mathbf{\Lambda}$ and offloading policy π, the revenue is defined as:

$$O(\mathbf{Q}(\mathbf{\Lambda},\pi),\mathbf{\Lambda}) = R_{\text{ow}}(\mathbf{Q}(\mathbf{\Lambda},\pi),\mathbf{\Lambda}) - L_{\text{ow}}(\mathbf{\Lambda},\pi). \tag{2.18}$$

Here

$$R_{\text{ow}}(\mathbf{Q}(\mathbf{\Lambda},\pi),\mathbf{\Lambda}) = \sum_{j\in\mathcal{J}} w^j \sum_{c\in\mathcal{C}} \lambda_c^j \sum_{k\in\mathcal{S}^j} \alpha_k^j q_{c,k}^j(\mathbf{\Lambda},\pi) \tag{2.19}$$

denotes the aggregate reward, and $L_{\text{ow}}(\mathbf{\Lambda},\pi)$ is the rate of computational work on users' devices associated with jobs that do not complete on time and with jobs that a user attempts to offload but ends up completing locally.

Weighted power savings with wastage revenue. We define it as a weighted sum of power savings with wastage per job type j. Given an offered load $\mathbf{\Lambda}$ and an offloading policy π, we calculate it as follows:

$$P(\mathbf{Q}(\mathbf{\Lambda},\pi),\mathbf{\Lambda}) = R_{\text{ps}}(\mathbf{Q}(\mathbf{\Lambda},\pi),\mathbf{\Lambda}) - L_{\text{ps}}(\mathbf{\Lambda},\pi) \tag{2.20}$$

[8] The weights w^j are used to prioritize one job type over another in the admission control policy. However, we still assume the underlying wireless scheduler is round-robin and unaware of job types.

where $R_{\text{ps}}(\cdot) = R_{\text{ow}}(\cdot)$ except that reward, α_k^j, is based on energy saved per job completion. $L(\mathbf{\Lambda}, \pi)$ represents the power expended at devices associated with jobs that miss their deadlines and jobs that a user attempts to offload but ends up executing locally.

2.4.1 Upper bound

Let $\mathbf{Q} = (\mathbf{q^1}, ..., \mathbf{q^J})$ be a vector of vectors, where $\mathbf{q^j} = (q_{c,k}^j : c \in \mathcal{C}, k \in \mathcal{S}^j)$, and $q_{c,k}^j$ is the fraction of type j jobs that belong to users with channel quality c, are cut at location k, and complete. We define

$$\begin{aligned} \Sigma = \{\mathbf{Q} \mid \mathbf{q}^j \geq \mathbf{0}, \sum_{k \in \mathcal{S}^j} q_{c,k}^j \leq 1 \ \forall \ j \in \mathcal{J}, \ c \in \mathcal{C}, \\ \text{and } q_{c,k}^j = 0 \ \forall \ j \in \mathcal{J}, \ c \in \mathcal{C}, \ k \in \mathcal{S}^j \setminus \mathcal{S}_c^j\} \end{aligned} \tag{2.21}$$

as the set of such possible vector of vectors. We then define the channel and edge server utilization per job type j based on $\mathbf{Q}$, which is long term fraction of completed jobs, as

$$\rho_{\text{ch}}^j(\mathbf{Q}) = \sum_{c \in \mathcal{C}} \lambda_c^j \sum_{k \in \mathcal{S}^j} q_{c,k}^j \frac{d_k^j}{r_c} \mathbf{1}(k \in \mathcal{S}_c^j), \tag{2.22}$$

$$\rho_{\text{ed}}^j(\mathbf{Q}) = \sum_{c \in \mathcal{C}} \lambda_c^j \sum_{k \in \mathcal{S}^j} q_{c,k}^j \frac{\gamma_k^j}{\omega} \mathbf{1}(k \in \mathcal{S}_c^j), \tag{2.23}$$

and let $\rho_{\text{ch}}(\mathbf{Q}) = \sum_{j \in \mathcal{J}} \rho_{\text{ch}}^j(\mathbf{Q})$ and $\rho_{\text{ed}}(\mathbf{Q}) = \sum_{j \in \mathcal{J}} \rho_{\text{ed}}^j(\mathbf{Q})$ denote the total channel and the total edge server utilization, respectively.

Recall that we defined $\mathcal{T}(\mathbf{\Lambda})$ to be the set of *feasible* long term fractions of successful job completion by stationary offloading policies when the system

load is $\mathbf{\Lambda}$. Here we define

$$\overline{\mathcal{T}}(\mathbf{\Lambda}) = \{\mathbf{Q} \mid \mathbf{Q} \in \Sigma,\ \rho_{\text{ch}}(\mathbf{Q}) \leq 1 \text{ and } \rho_{\text{ed}}(\mathbf{Q}) \leq 1\} \tag{2.24}$$

as a natural outer bound.

Theorem 2. Given an offered load $\mathbf{\Lambda}$ we have that $\mathcal{T}(\mathbf{\Lambda}) \subseteq \overline{\mathcal{T}}(\mathbf{\Lambda})$. Then the maximum weighted net timely offloaded workload revenue achievable by any stationary offloading policy is defined as

$$O^*(\mathbf{\Lambda}) := \max_{\mathbf{Q} \in \mathcal{T}(\mathbf{\Lambda})} O(\mathbf{Q}, \mathbf{\Lambda}) \leq \max_{\mathbf{Q} \in \overline{\mathcal{T}}(\mathbf{\Lambda})} R_{\text{ow}}(\mathbf{Q}, \mathbf{\Lambda}) \tag{2.25}$$

Similarly, the maximum weighted power savings with wastage revenue achievable by any stationary offloading policy is defined as

$$P^*(\mathbf{\Lambda}) := \max_{\mathbf{Q} \in \mathcal{T}(\mathbf{\Lambda})} P(\mathbf{Q}, \mathbf{\Lambda}) \leq \max_{\mathbf{Q} \in \overline{\mathcal{T}}(\mathbf{\Lambda})} R_{\text{ps}}(\mathbf{Q}, \mathbf{\Lambda}) \tag{2.26}$$

Proof. Similar to proof of Theorem 1. □

Remark 2. We let $\mathbf{Q}^*_{\text{ow}}(\mathbf{\Lambda}) = \text{argmax}_{\mathbf{Q} \in \overline{\mathcal{T}}(\mathbf{\Lambda})} R_{\text{ow}}(\mathbf{Q}, \mathbf{\Lambda})$ and $\mathbf{Q}^*_{\text{ps}}(\mathbf{\Lambda}) = \text{argmax}_{\mathbf{Q} \in \overline{\mathcal{T}}(\mathbf{\Lambda})} R_{\text{ps}}(\mathbf{Q}, \mathbf{\Lambda})$ denote the maximizers associated with the bounds for the two reward models. Once again, for brevity we will use $\mathbf{Q}^*$.

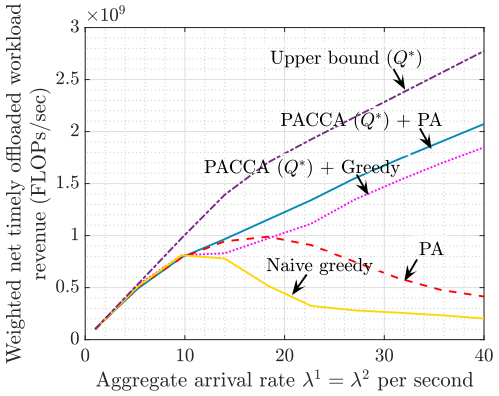
2.4.2 Simulation results

In this subsection, we compare the performance of various offloading management policies when dealing with heterogeneous jobs. For PACCA, we determine the admission control probabilities per combination of job types,

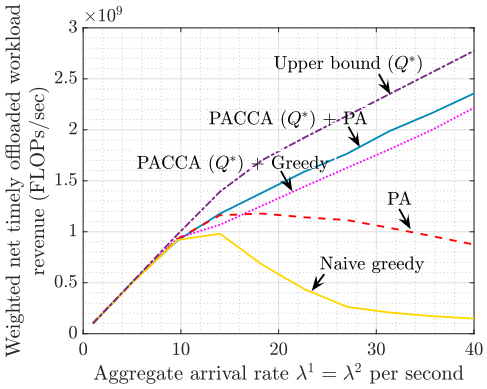
user channel qualities, and cut location based on the fractions that maximize system revenue, i.e., $\mathbf{Q}^*$. We then again schedule the offloading of admitted jobs using either Greedy or PA policies, i.e., PACCA + Greedy or PACCA + PA. Due to space constraints, we omitted our study of the robustness of PACCA + PA for the heterogeneous case.

Setting. The simulation involves 20 users generating two job types, AlexNet and DeepFace. Half of the users (i.e., 10) generate job Type 1, while the other half generates job Type 2. Each user in either category generates jobs at an equal rate per second based on a Poisson distribution with intensity $\lambda^j/10$, where λ^j is the aggregate arrival rate for job type j. We have set equal aggregate arrival rate for the two job types. Users for each job type are divided equally into two channel quality groups, with half (i.e., 5) offloading over channel Quality 1 and the other half over channel Quality 2. For more information on the simulation parameters, refer to Table 2.1 and Fig. 2.2.

Results discussion. Figures 2.11 and 2.12 illustrate the weighted net timely offloaded workload and weighted power savings with wastage revenue achieved by various offloading policies, respectively. As observed in the case of homogeneous jobs, the PACCA + PA policy outperforms other policies. However, the relative performance of PA policy has declined compared to the homogeneous case since it only considers the residual data and channel capacity, ignoring a job's weight relative to others. In contrast, PACCA coordinates job admission control and cut assignment based on how much relative revenue each job type and cut will generate.

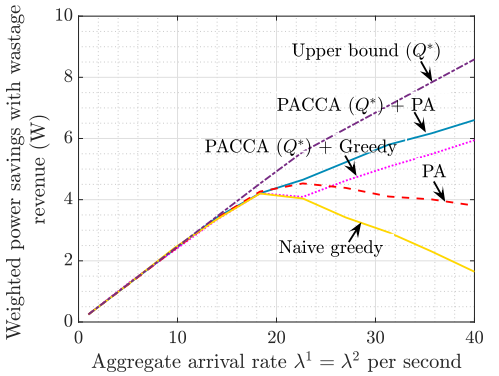


(a) Strict delay deadline, $(\tau^1, \tau^2) = (0.4\tau^1_{\max}, 0.8\tau^2_{\max})$.

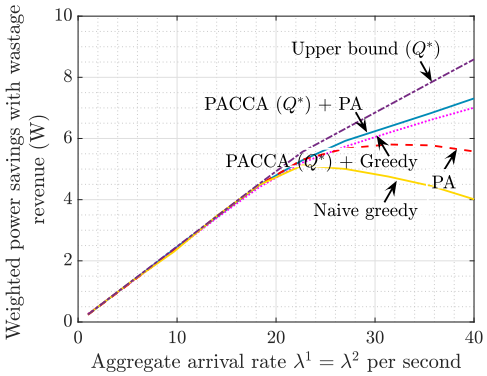


(b) Relaxed delay deadline, $(\tau^1, \tau^2) = (0.8\tau^1_{\max}, 0.8\tau^2_{\max})$.

Figure 2.11: Comparing the weighted net timely offloaded workload revenue for different policies with $(w^1, w^2) = (0.97, 0.03)$.



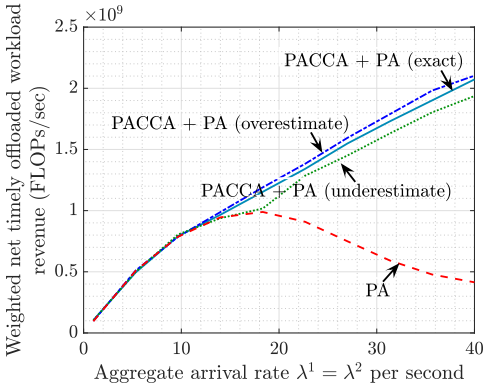
(a) Strict delay deadline, $(\tau^1, \tau^2) = (0.4\tau^1_{\max}, 0.8\tau^2_{\max})$.



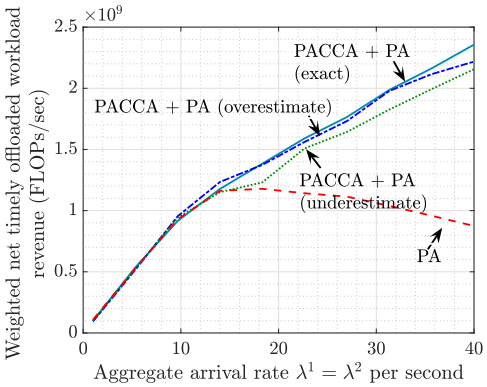
(b) Relaxed delay deadline, $(\tau^1, \tau^2) = (0.8\tau^1_{\max}, 0.8\tau^2_{\max})$.

Figure 2.12: Comparing the weighted power savings with wastage revenue for different policies with $(w^1, w^2) = (0.97, 0.03)$.

Robustness study: We evaluate the robustness of PACCA + PA to imperfect estimates of the offered job loads in Figures 2.13a and 2.13b, for the weighted net timely offloaded workload revenue under a strict and relaxed deadline respectively. Following a similar approach to the homogeneous case, we simulated perturbed arrival rates of jobs per user that affect the aggregate load per channel and job type. We optimize PACCA + PA for the corresponding loads and compare three scenarios: (i) PACCA + PA (exact), where we provide PACCA with the exact loads, i.e., λ^j; (ii) PACCA + PA (overestimate), where we provide PACCA with an overestimate of the loads, i.e., $\lambda^j \cdot (1 + x\%)$, leading to less offloading for the lower weighted jobs as compared to (i) in the considered regimes; and (iii) PACCA + PA (underestimate), where we provide PACCA with an underestimate of the loads, i.e., $\lambda^j \cdot (1 - x\%)$, resulting in more offloading for both job types as compared to (i). We show PA as a baseline. The results show that for an estimation error of x equal to 25%, PACCA + PA (overestimate) is almost indistinguishable from PACCA + PA (exact). This is because it accepts fewer lower-weighted jobs, which do not significantly contribute to the total revenue. However, with underestimation, PACCA + PA (underestimate) has identical performance to PA for certain loads, resulting in a performance gap of 11% as compared to PACCA + PA (exact) in the worst case. We see similar trends with the weighted power savings with wastage revenue in Fig. 2.14.



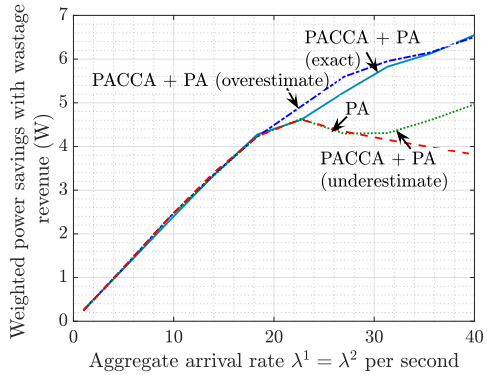
(a) Strict delay deadline, $(\tau^1, \tau^2) = (0.4\tau^1_{\max}, 0.8\tau^2_{\max})$.



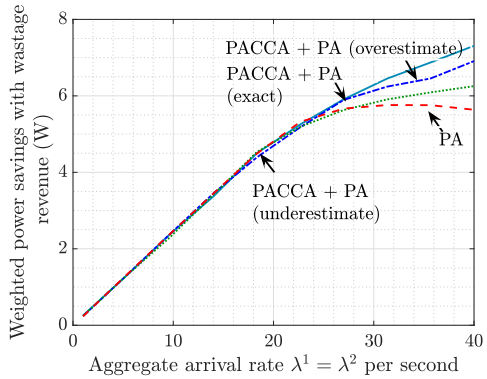
(b) Relaxed delay deadline, $(\tau^1, \tau^2) = (0.8\tau^1_{\max}, 0.8\tau^2_{\max})$.

Figure 2.13: Evaluating PACCA + PA's robustness with $(w^1, w^2) = (0.97, 0.03)$ and 25% deviation from exact load knowledge with strict vs. relaxed deadline for weighted net timely offloaded workload revenue

.



(a) Strict delay deadline, $(\tau^1, \tau^2) = (0.4\tau^1_{\max}, 0.8\tau^2_{\max})$.



(b) Relaxed delay deadline, $(\tau^1, \tau^2) = (0.8\tau^1_{\max}, 0.8\tau^2_{\max})$.

Figure 2.14: Evaluating PACCA + PA's robustness with $(w^1, w^2) = (0.97, 0.03)$ and 25% deviation from exact load knowledge with strict vs. relaxed deadline for weighted power savings with wastage revenue

.

2.5 Conclusion

Managing heterogeneous compute job offloading in the MEC fabric subject to delay constraints presents significant challenges that require careful management of user device, channel, and edge server resources while considering different job characteristics and system loads. To address this, we have detailed a comprehensive framework, which relies on job profiling, using probabilistic admission control and cut assignment, coupled with a predictive abandonment policy that abandons offloads unlikely to meet their deadline (this not only frees up resources for jobs with more promise but also avoids throughput collapse). Our proposed approach, PACCA + PA, is expected to perform robustly and effectively but requires prior knowledge of offered job rates across job types and channel qualities. If this is not known, signal processing techniques such as window averaging can be employed to learn the offered job rate over time.

Chapter 3

Fundamentals of Caching Layered Data Objects

This chapter [1] looks at caching data that comes in versions. In a multiple-representation (MR), every version is separate. In a layered representation (LR), a high-quality version is just the low-quality one plus a few extra layers.

We check how familiar rules like LRU, LFU, and Belady behave under both representations. With a simple working-set model we show when LR wins, when MR wins, and why adding more layers is not always a good idea—it helps only if the extra layers are small enough and requested often enough to justify their space in the cache.

[1] This chapter is based on the published work in [7]:

- A. Bari, G. De Veciana, G.Kesidis, "Fundamentals of Caching Layered Data Objects," *To appear in ICDCS '25*

Agrim Bari led the formulation of the problem, the design of policies, execution of experiments, and the writing of the paper.

3.1 Introduction

Managing shared edge caching. Efficient management of shared memory systems for applications requiring large amounts of data continues to be a challenging problem. These challenges are exacerbated when mobile applications with latency constraints leverage limited/costly edge caching resources but have limited or variable connectivity to the network edge. In such settings, ensuring data is available when needed is all the more critical.

Layered representations and applications. The focus of this chapter is on applications where data objects can be stored and be of use in multiple versions which are encoded in Layered Representations (LRs). Each version of a data object embodies a tradeoff in size, and thus resource requirements, versus the 'quality' that an application can extract. LRs are such that the cumulative availability of each additional layer delivers a version with improved quality. Such an incremental approach to representing data object versions brings flexibility to systems where applications have heterogeneous quality requirements or can tolerate quality degradation when resources are scarce. LRs have found applications in, e.g., zoomable maps, video compression, and Virtual Reality (VR) games. For maps, LRs can be used to deliver different levels of topographic detail. Similarly, scalable video coding includes a base layer that contains essential information for lower quality, while enhancement layers contribute details for higher quality. In computer graphics, particularly relevant to VR, progressive meshes [25] can be used for efficient storage and rendering of 3D models. Through iterative mesh simplification

algorithms, along with vertex splitting and collapse operations, a hierarchical structure emerges. Each level in this hierarchy represents a progressively more detailed version of the original mesh. LRs are also applicable when seeking to have compact Neural Network (NN) models. In this context, the base NN model can deliver lower inference accuracy but can be enhanced through additional data layers, which increase model complexity or weight fidelity, resulting in an NN model with higher inference accuracy [33, 37].

Exploiting layered representations. Applications may request different versions of a data object for various reasons. First, an application might consider the computational resources of the end device, e.g., an end device with limited resources may request a version that requires less compute/memory/energy. Second, data object requests may vary based on the communication network conditions. In instances where the end device has limited bandwidth, preventing it from receiving high-quality data promptly, users may opt for versions that balance quality with efficient transmission. Thirdly, an application may simply not require the highest quality version. For example, in VR gaming settings, a detailed model for a complex tree that is far away would not be visible and thus is not required. Overall, these diverse considerations highlight the needed flexibility that LRs would be able to satisfy for a range of application requirements and constraints.

Alternative representations. There are other ways of representing different versions of data objects, e.g., Multiple Representations (MRs) with or without transcoding, see [21, 54, 56]. In the case of MR without transcoding,

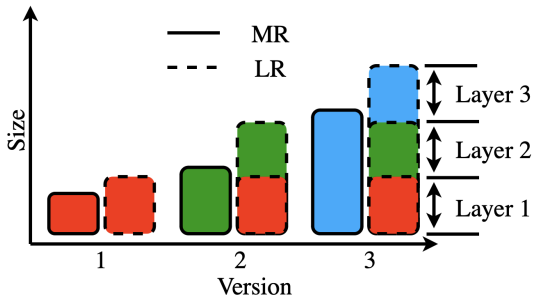


Figure 3.1: 3 Versions under Multiple/Layered representation of a data object.

discrete and independent versions of data objects are created — in the context of video streaming, these versions correspond to encoding video at different rates without any layering. MR may be more storage-efficient as compared to LR for the same version because LR may require additional information to extend to higher versions, see Fig. 3.1. However, LR may be more storage efficient when plural quality/resolution levels of the same object are simultaneously in demand because the MR version will have a significant amount of identical (lower quality) information.

Whereas, MR with transcoding involves storing only one version corresponding to the highest 'quality' level - so, any lower version can be readily computed from this version. This transcoding can occur either in real-time (online transcoding), generating lower versions in real-time upon request, or in non-real-time (offline transcoding) where transcoding takes place in the backend for potential future requests. However, in this chapter, we will focus solely on MR without transcoding limiting the computational burdens and delays associated with real-time transcoding.

Caching policies with layered representations. While considerable attention has been paid to the design and analysis of caching policies for MR, limited attention has been given to the LR setting which as mentioned above, we expect will be of increasing relevance to emerging applications and caching at the network edge. In this chapter, we focus on a disciplined study of traditional caching policies which have been redesigned to leverage LR data representations. Below we briefly discuss relevant related research before summarizing our contributions.

3.1.1 Related work

Analytical works on caching policies and approximation. We restrict our review to papers most relevant to our work. [11, 29] summarize significant early work in the design of caching policies, and [22] describes analytical methods and evaluation results for the performance assessment of caching strategies. The aim of any caching policy is to achieve efficient cache utilization. This efficiency is measured primarily by the cache hit rate, which is the averaged fraction of data object requests for which the data object is in the cache when requested.

Besides hit rate, other design objectives for caching policies are ease of implementation, low operational overhead, and adaptability to fluctuations in access/request patterns. An important difference among caching policies is in what they evict when the cache is full. Under Least Recently Used (LRU), the cache is consistently updated to hold the most recently requested data

objects, enabling it to leverage the temporal locality of data object requests. Notably for LRU under the Independent Reference Model (IRM), where each data object is requested independently of any past requests, the invariant distribution assuming data objects of the same size [38] and an approximation for the hit rate [12, 15, 9, 20] have been obtained. In particular, [12] describes the working-set approximation for hitting probabilities, the fraction of requests for a data object for which the object is in the cache. This approximation has been shown to be accurate as the number of objects scales [15, 20]. We herein extend the analysis of [12] for the approximation to our setting, where data objects have layered representations, and also demonstrate the asymptotic accuracy of the approximation based on ideas from [15].

Under the IRM model, for a fixed cache capacity with same-size data objects, caching the most popular data objects is optimal for causal policies [2]. Least Frequently Used (LFU) performs optimally under stationary regimes of request patterns by replacing cached data objects based on the frequency measurements of past requests. An interesting work by [28] shows that a variant of LRU that infers the instantaneous request rate subject to the history of requests can come arbitrarily close to the optimal LFU algorithm. [30] shows that even for strongly correlated request patterns, LFU is still optimal among causal policies. However, while LFU may be effective in stationary scenarios where access patterns remain relatively constant, it may struggle to perform optimally in non-stationary regimes where the dynamics of data access change over time.

Moving beyond the IRM, various researchers have conducted competitive analyses, considering the total number of cache misses as the figure of merit [58, 19]. They compare the performance of an online policy, i.e., one that makes eviction decisions without knowledge of future requests, with the optimal offline policy, Belady [8], i.e., one that knows the entire sequence of requests in advance. The LRU policy has a competitive ratio that scales linearly with the cache size, B. Improving on LRU, researchers in [19] have shown that a randomized online algorithm, Marker, which uses markers to decide and prioritize critical data objects, could be worse than the optimal offline algorithm by a factor of $2H_B \approx 2\log(B)$ (H_B denotes the Bth harmonic number: $H_B = 1 + 1/2 + 1/3 + \ldots + 1/B$), but not more. Moreover, no online algorithm could achieve a factor less than H_B. Recently, [47] extended these results to cases where the traditional marker algorithm is combined with predictions about the next time of request for objects currently in the cache when making eviction decisions. If done correctly, they show that one can improve upon this factor of $\log(B)$ depending on the accuracy of the prediction.

Multiple vs. Layered Representations. Researchers have also explored the caching problem for objects which could either be in MR or LR. In [21], the authors advocate for storing MR for some data objects and LR for the rest when the goal is to maximize the hit rate. In addition, they also develop heuristic policies that dynamically adapt the representation for each data object. [55] compares optimization-based static caching policies for MR versus LR to conclude that the hit rate for an LR based caching policy is

superior. We address this question more broadly by showing the benefit of LR over MR in terms of hit rate as a function of MR's storage efficiency compared to LR. We also present results showing the benefit of LR as we vary the cache size, the fraction of requests for different versions, the relative size of versions, and the number of versions.

3.1.2 Contributions and organization

The main contributions of this chapter are now summarized. First, we redesign and analyze traditional caching policies (Belady, LFU, LRU) for settings where data objects are available in MR or LR. In particular, we introduce a new working set approximation to compute the hit probabilities for data objects in a cache utilizing Layered LRU (LLRU) caching policy under an IRM for requests for data object versions. We show the asymptotic accuracy of this approximation for both a fixed number of layers and a continuum of layered representations. The continuum model seems appropriate for settings where layering overhead is minimal and thus applications could in principle cache only the data it requires for where it needs better quality, e.g., in a VR gaming setting where high quality is needed only for aspects of the environment that are currently (or maybe in the near future) close by.

Second, using the working-set approximation, we evaluate the benefit of LR versus MR for a fixed set of equivalent data object versions. Our results suggest that even if LR incurs relatively high overheads versus MR, the performance benefits of LR representations are excellent. We note however

that this does depend on the popularity of the distinct versions, i.e., the layered structure is particularly beneficial when there is sufficient diversity in the requests for a data object's versions. With these observations in mind, we consider greedy caching policies that might exploit the availability of *both* LR and MR, by greedily seeking to represent the versions in the cache in the most memory-efficient manner. Such policies can provide some benefit but only under highly skewed popularity for data object versions. We also explore the performance of various layered caching policies under stationary IRM showing that layered LRU is not quite on par either with layered LFU or, of course, the genie-based layered Belady; yet LLRU can be expected to be a workhorse for caching LR based systems because of its simplicity and robustness to dynamic request distribution.

Finally, we explore the performance sensitivity of LLRU to the size and popularity of layers and data object versions. This provides an avenue to study how many layers are enough or when indeed more layers leads to better performance.

The chapter is organized as follows. We start by describing the system model and working-set approximation for the LLRU policy in Section 3.2. In the same section, we describe the re-design of traditional caching policies (Belady, LFU, LRU) with LRs along with an optimization-based static-offline caching policy. In Section 3.3 we empirically evaluate the claims of Section 3.2. Finally, Section 3.4 concludes the chapter.

3.2 System Model and analysis

3.2.1 Model for cache

The system consists of a cache server of capacity B bytes. The server stores various versions of data objects to serve near-future requests from a user population. Owing to practical constraints, the cache capacity typically is not enough to store all versions of data objects.

3.2.2 Model for data objects and arrival requests

We let $\mathcal{D}$ denote the set of these data objects - the set has cardinality $D = |\mathcal{D}|$. Each data object $d \in \mathcal{D}$ can be stored in several versions, $v \in \{1, 2, \ldots, V\}$, where V is the number of versions. We adopt the Independent Reference Model (IRM), which is a good abstraction for independent requests generated from a large population of users. Let $\lambda(d, v)$ denote the arrival rate of requests for version v of data object d. The total arrival rate of requests for data object $d \in \mathcal{D}$ is given by $\lambda(d) = \sum_{v=1}^{V} \lambda(d, v)$ and $\lambda = \sum_{d=1}^{D} \lambda(d)$ denotes the total arrival rate of requests generated by a population of users. We denote the vector of requests for each version and data object as $\boldsymbol{\lambda} = (\lambda(d, v) : d \in \mathcal{D}, v \in \{1, 2, \ldots, V\})$. We define $q(d) = \lambda(d)/\lambda$ and $q(d, v) = \lambda(d, v)/\lambda$ as the probability of request for data object $d \in \mathcal{D}$ and probability of request for data object $d \in \mathcal{D}$ in version v, respectively. We consider two representations for storing these data object versions as explained below.

3.2.3 Model for Multiple Representations

Under *Multiple Representations* (MRs), several distinct versions of a data object can be maintained in the cache. Let $s_{\mathrm{MR}}(d, v)$ denote the cache storage space occupied by data object $d \in \mathcal{D}$ in version v under MR. The size of versions of a data object d under MR is strictly increasing in v, i.e., $s_{\mathrm{MR}}(d, 1) < s_{\mathrm{MR}}(d, 2) < \cdots < s_{\mathrm{MR}}(d, V)$. As explained before, if one version of a data object is cached and there is a request for a different version of the same data object, the cached version cannot be used to serve this request under multiple representations.

3.2.4 Model for Layered representations

Under *Layered Representations* (LRs), a version v of a data object is represented by a set of consecutive layers $l \in \{1, 2, \ldots, v\}$ where the size of layer l for data object d is denoted by $\delta(d, l)$. So version v of data object d occupies $s_{\mathrm{LR}}(d, v) = \sum_{l=1}^{v} \delta(d, l)$ space in the cache. Note that the incremental layer sizes $\delta(d, l)$ need not be strictly increasing or decreasing in l. We will be exploring the impact of this in later sections. Depending on the application, we expect the overall size of representations under LR to be larger than MR, i.e., $s_{\mathrm{MR}}(d, v) \leq s_{\mathrm{LR}}(d, v)$. We will be studying the impact of such overheads in the sequel. We let $\gamma(d, l) = \sum_{v=l}^{V} \lambda(d, v)$ denote the total request rate for layer l induced by the requests for different versions of data object d and $p(d, l) = \gamma(d, l)/\lambda = \sum_{v=l}^{V} q(d, v)$ denote the request probability for layer l of data object d.

3.2.5 Caching Policies

We consider a set Π of caching policies. These policies can either be online or offline, i.e., they adapt the cached content based on incoming requests or not, respectively, and they may have knowledge about the future requests or request rate. For a given vector of request rate $\boldsymbol{\lambda}$ and policy $\pi \in \Pi$, we define $\mathbf{h}_{\boldsymbol{\lambda},\pi} = (h_{\boldsymbol{\lambda},\pi}(d,r) : d \in \mathcal{D}, v \in \{1,2,\ldots,V\})$, where $h_{\boldsymbol{\lambda},\pi}(d,v)$ denotes the long-term fraction of requests for data object d and version v that results in a cache hit. These data objects could either be stored in LR or MR.

3.2.6 Performance metric

We capture the overall performance of the cache in terms of hit rate. For a given vector of request rates $\boldsymbol{\lambda}$ under policy π, we define it as

$$H_{\boldsymbol{\lambda},\pi} = \sum_{d=1}^{D} \sum_{v=1}^{V} \lambda(d,v) h_{\boldsymbol{\lambda},\pi}(d,v). \tag{3.1}$$

3.2.7 Layered Caching Policies

We now introduce our caching policies for layered representations. Note that for all policies, a user request to access an object involves using a hash table to determine whether the object is cached (i.e., whether it's a cache hit) and, if so, where it is stored in cache memory. We first define a common property of all layered caching policies stated hereafter.

Property of layered caching policies. For all policies discussed hereafter, if layer $l+1$ is present in the cache, then layer $i \in \{1,2,\ldots,l\}$ is

also present in the cache.

3.2.7.1 Static optimal

We begin by developing an optimization-based static-caching policy that maximizes the hit rate *given* the vector of request rates $\boldsymbol{\lambda}$, where data objects are in LRs. This is the best that a policy with no knowledge of future requests can do. Let $\mathbf{x} = (x(d,v) : d \in \mathcal{D}, v \in \{1,2,\ldots,V\})$, where $x(d,v)$ denotes an indicator for whether data object d in version v is included in the cache or not. We formulate the following optimization that maximizes the hit rate.

$$\max_{\mathbf{x}} \quad \sum_{d=1}^{D}\sum_{v=1}^{V} \lambda(d,v)x(d,v) \tag{3.2a}$$

$$\text{s.t.} \quad \sum_{d=1}^{D}\sum_{v=1}^{V} \delta(d,v)x(d,v) \leq B, \tag{3.2b}$$

$$x(d,v-1) \geq x(d,v), \quad \forall\, d \in \mathcal{D}, v \in \{2,3,\ldots,V\}, \tag{3.2c}$$

$$x(d,v) \in \{0,1\} \qquad \forall\, d \in \mathcal{D}, v \in \{1,2,\ldots,V\} \tag{3.2d}$$

where Constraint 4.20b is on the cache capacity and Constraint 4.20c is ensuring the previously mentioned property of layered caching policies.

3.2.7.2 Layered Least Frequently Used (LLFU)

The LLFU caching policy prioritizes the caching of layers of data objects that have been accessed most frequently while ensuring Constraint 4.20c.

Although we will show through simulations that this policy is optimal when the size of each layer is equal among policies with no knowledge of future requests, it involves tracking and updating access frequencies of each layer of every data object. Thus, it may not be practical.

An LLFU cache serves an incoming request for object (d, v) as follows:

- The number of accesses for layer (d, l) is incremented for all $l \leq v$.
- If a cache miss occurs (i.e., layer (d, v) is not present in the cache), to meet Constraint 4.20b, the server may need to evict layers of cached data objects in increasing order of their current number of accesses until there is enough space to store all layers $l \leq v$ for data object d.

So, under LLFU, only layers with the currently highest access counts are cached. Periodically to prevent numerical overflow, access counts *of all data object layers* can, e.g., be simultaneously decremented by a common amount (equal to the currently smallest access count among all data object layers). Instead of access counts, one can define an LLFU policy with access frequencies equal to the inverse of auto-regressive estimates of inter-access times.

A hybrid LR-MR LFU policy is described in Section 3.2.10 below.

3.2.7.3 Layered Least Recently Used (LLRU)

LLRU manages the cache by evicting the least recently accessed layers among all data objects currently in the cache. This policy works well when

there is temporal locality of request patterns, i.e., layers and data objects accessed more recently are more likely to be accessed again in the near future. Let $a(d, l)$ denote the time of last access for layer l and data object $d \in \mathcal{D}$. An LLRU cache serves an incoming request for (d, v) at time t as follows:

- Set $a(d, l) = t$ for all $l \leq v$ while ensuring that lower layers come after the higher layers.
- If a cache miss occurs, to meet Constraint 4.20b the server may need to evict layers of cached data objects in the increasing order of access times until there is enough space to store all layers $l \leq v$ for data object d.

Instead of using access times, the LLRU cache-eviction order can be maintained by just using a doubly-linked list.

A hybrid LR-MR LRU policy is described in Section 3.2.10 below.

3.2.7.4 Layered Belady (LBelady)

LBelady evicts by identifying layers of data objects that will be accessed furthest in the future and is thus a non-causal policy. Let $f(d, l; t) > t$ denote the smallest access time after t for layer l of data object d. When the size of each layer is equal, this is the optimal policy among all possible policies, albeit accurate future knowledge is generally not available. In our simulations, this will serve as a benchmark for the case of equal layer sizes.

An LBelady cache serves an incoming request at time t in the following manner:

- If a cache miss occurs, to meet Constraint 4.20b, the server may evict layers of data objects (d, l) in decreasing order of $f(d, l; t)$ until there is enough space to store the requested version.

3.2.8 Working-set approximation for LLRU

We present a working-set approximation for the LLRU policy when data objects are stored in layered representations. We will demonstrate its accuracy by studying this working set as number of data objects go to infinity in the next section and in a later section through simulations.

Consider a system where time is divided into slots. For the analysis we assume that the request arrival process for each data object d and layer l is a Bernoulli process with parameter $p(d, l)$, i.e., the probability that there is a request for data object d and layer l in a time slot is $p(d, l)$ and such events occur independently across time slots.

3.2.8.1 Characteristic time

Suppose there is a request for data object d and layer l at time zero. Let $T_f(i, k)$ be the time of first request for data object $i \neq d$ and layer k, where $k = \{1, 2, \ldots, V\}$. We use $T_n(d, m)$ to denote the time of next query for data object d and layer m, where $m \leq l$. Under the Bernoulli arrival process model, these times are geometrically distributed, i.e., $T_f(d, l) \sim \text{Geo}(p(d, l))$ or $T_n(d, l) \sim \text{Geo}(p(d, l))$.

At time $t > 0$, the total size of different data objects and layer requested

up to time t (i.e., working-set size), excluding requests for data object d and layer l is given by:

$$S_{-(d,l)}(t) = \sum_{\substack{i=1 \\ i \neq d}}^{D} \sum_{k=1}^{V} \delta(i,k) \mathbf{1}\left\{T_f(i,k) < t\right\} + \sum_{k=1}^{l-1} \delta(d,k) \mathbf{1}\left\{T_n(d,k) < t\right\}, \quad (3.3)$$

where $\delta(i,k)$ represents the size of layer k for data object i.

The characteristic time $T_{-(d,l)}(B)$, a random variable, is defined as the minimum time $t > 0$ at which the working-set size excluding data object d and layer l exceeds B:

$$T_{-(d,l)}(B) = \min\{t > 0 : S_{-(d,l)}(t) \geq B\}. \quad (3.4)$$

A request for data object d and layer l at time $T_n(d,l)$ is a cache hit if the working-set size remains below B, i.e., $S_{-(d,l)}(T_n(d,l)) < B$, or equivalently, if $T_n(d,l) < T_{-(d,l)}(B)$. This relationship is expressed as:

$$\{S_{-(d,l)}(T_n(d,l)) < B\} = \{T_{-(d,l)}(B) > T_n(d,l)\}. \quad (3.5)$$

Thus, the hit probability for data object d and layer l is then

$$h(d,l) = \mathbb{P}\left(T_{-(d,l)}(B) > T_n(d,l)\right) = \mathbb{E}\left[1 - (1 - p(d,l))^{\left(T_{-(d,l)}(B)-1\right)}\right]. \quad (3.6)$$

Since $T_{-(d,l)}(B)$ corresponds to the time when the working-set size first reaches B, we have:

$$B = \sum_{\substack{i=1 \\ i \neq d}}^{D} \sum_{k=1}^{V} \delta(i,k) \mathbf{1}\left\{T_f(i,k) < t\right\} + \sum_{k=1}^{l-1} \delta(d,k) \mathbf{1}\left\{T_n(d,k) < t\right\}. \quad (3.7)$$

and taking expectations on both sides and simplifying,

$$B = \sum_{\substack{i=1 \\ i \neq d}}^{D} \sum_{k=1}^{V} \delta(i,k) \mathbb{E}\left[1 - (1 - p(i,k))^{T_{-(d,l)}(B)-1}\right] \quad + \\ \sum_{k=1}^{l-1} \delta(d,k) \mathbb{E}\left[1 - (1 - p(d,k))^{T_{-(d,l)}(B)-1}\right]. \quad (3.8)$$

We use two common approximations from the literature to simplify hit probability calculations; see [9, 20] for details.

Approximation 1: For $D \gg 1$, the characteristic time $T_{-(d,l)}(B)$ becomes concentrated around its mean value. Therefore, $T_{-(d,l)}(B)$ can be approximated by a deterministic value $t_{-(d,l)}(B)$ for data object d and layer l. Thus, the above equation can be rewritten as follows:

$$B = \sum_{\substack{i=1 \\ i \neq d}}^{D} \sum_{k=1}^{V} \delta(i,k) \left(1 - (1 - p(i,k))^{t_{-(d,l)}(B)-1}\right) \quad + \\ \sum_{k=1}^{l-1} \delta(d,k) \left(1 - (1 - p(d,k))^{t_{-(d,l)}(B)-1}\right). \quad (3.9)$$

The above is a fixed point equation, which can solved to find $t_{-(d,l)}(B)$ and one can use that to find the hit probability for data object d and layer l by

$$h(d,l) = \left(1 - (1 - p(d,l))^{\left(t_{-(d,l)}(B)-1\right)}\right). \quad (3.10)$$

Approximation 2: The dependence of $t_{-(d,l)}(B)$ on (d,l) can be ignored for all data objects and layer. This is works when $p(d,l)$ is relatively insignificant to

1, and becomes exact if request probabilities are equiprobable. In summary, the working-set approximation for LLRU is as follows. Let $t^*(B)$ be such that:

$$B = \sum_{d=1}^{D} \sum_{l=1}^{V} \delta(d,l) \left(1 - (1 - p(d,l))^{(t^*(B)-1)}\right) \tag{3.11}$$

Then the hit probability for data object $d \in \mathcal{D}$ and layer $l \in \{1, 2, \ldots, V\}$ is given by

$$h(d,l) = \left(1 - (1 - p(d,l))^{(t^*(B)-1)}\right). \tag{3.12}$$

This hit probability for data object d and layer l is equal to hit probability for data object d and version v, where $v = l$ because of the property of LLRU. The results for a time-slotted system can be extended to continuous time, where the request arrival process for data object d and layer l is a Poisson process with parameter $\gamma_{d,l}$. The hit probability for data object d and layer l is given by

$$h(d,l) = 1 - e^{-\gamma_{d,l} t^*(B)}, \tag{3.13}$$

where $t^*(B)$ is such that:

$$B = \sum_{d=1}^{D} \sum_{l=1}^{V} \delta(d,l) \left(1 - e^{-\gamma_{d,l} t^*(B)}\right). \tag{3.14}$$

In the next section, we show the asymptotic accuracy of working-set approximation.

3.2.9 Asymptotic accuracy of working-set approximation for LLRU

We extend the analysis from [15] to incorporate layers into the construction, focusing on LR for this part. Specifically, we consider a system of

caches where the request probability for data objects and the working-set size scale as a function of D. In this framework, each data object is assumed to have V fixed layers (or versions).

Let F be a smooth, monotone increasing function with domain [0, 1], such that $F(0) = 0$ and $F(1) = 1$. We define the request probability for data object d and version v as D scales in the following manner:

$$q^{(D)}(d, v) = (F(d/D) - F((d-1)/D))\, g(v; d/D) \tag{3.15}$$

where $g(v; d/D)$ denotes the request probability for version v of data object d and $\sum_{v=1}^{V} g(v; d/D) = 1$ for all data objects. Based on the definition of F and g, we have $\sum_{d=1}^{D} \sum_{l=1}^{V} q^{(D)}(d, v) = 1$ and $q^{(D)}(d, v) \geq 0$. Thus, $q^{(D)}(d, v)$ is a probability distribution determined by F and g. We use $\delta^{(D)}(d, l)$ to denote the size of layer l for data object d and $p^{(D)}(d, l) = \sum_{v=l}^{V} q^{(D)}(d, v)$ denotes the request probability for layer l of data object d.

We define $b = B/D$, which scales as a function of D, and develop the notion of characteristic time in the same way as in the previous section. We assume a system with time-slots and request arrival process for data object d and layer l is a Bernoulli process with parameter $p^{(D)}(d, l)$ and such events occur independently over time-slots.

Suppose there is a request for data object d and layer l at time zero. Let $T_f^{(D)}(i, k)$ be the time of first request for data object $i \neq d$ and layer k, where $k = \{1, 2, \ldots, V\}$. We use $T_n^{(D)}(d, m)$ to denote the time of next query for data object d and layer m, where $m \leq l$. Under the Bernoulli arrival

process model, these times are geometrically distributed, i.e., $T_f^{(D)}(d,l) \sim \text{Geo}(p^{(D)}(d,l))$ or $T_n^{(D)}(d,l) \sim \text{Geo}(p^{(D)}(d,l))$. At time $t > 0$, the total size of different data objects and layer requested upto time t (i.e., working-set size), excluding requests for data object d and layer l is

$$S_{-(d,l)}^{(D)}(t) = \sum_{k=1}^{l-1} \delta^{(D)}(d,k)\mathbf{1}\left\{T_n^{(D)}(d,k) < t\right\} \quad + \sum_{\substack{i=1 \\ i\neq d}}^{D}\sum_{k=1}^{V} \delta^{(D)}(i,k)\mathbf{1}\left\{T_f^{(D)}(i,k) < t\right\}, \quad (3.16)$$

with

$$\mathbb{E}\left[S_{-(d,l)}^{(D)}(t)\right] = \sum_{k=1}^{l-1} \delta^{(D)}(d,k)\left(1-(1-p^{(D)}(d,k))^{(t-1)}\right) \quad + \sum_{\substack{i=1 \\ i\neq d}}^{D}\sum_{k=1}^{V} \delta^{(D)}(i,k)\left(1-(1-p^{(D)}(i,k))^{(t-1)}\right). \quad (3.17)$$

Similarly, we can find the working-set size at time t and its expectation is given by:

$$\mathbb{E}\left[S^{(D)}(t)\right] = \sum_{d=1}^{D}\sum_{l=1}^{V} \delta^{(D)}(d,l)\left(1-(1-p^{(D)}(d,l))^{(t-1)}\right). \quad (3.18)$$

We define Riemann integrable Δ satisfying $\Delta(d/D, l) = \delta^{(D)}(d,l)$ for all D, d and l for the theorem below.

Theorem 3.1 (Asymptotic hit probability)**.** *Consider the system of caches which scales as a function of D. For large D, the hit probability for data object d and layer l, $h^{(D)}(d,l)$, is approximated by*

$$h^{(D)}(d,l) = \left(1-(1-p^{(D)}(d,l))^{(t^*(B)-1)}\right) \quad (3.19)$$

where $t^*(B)$ is such that:

$$B = \sum_{d=1}^{D} \sum_{l=1}^{V} \delta^{(D)}(d,l) \left(1 - (1 - p^{(D)}(d,l))^{(t^*(B)-1)}\right). \tag{3.20}$$

In the limit $D \to \infty$, the hit probability for data object d and layer l is given by:

$$h(d,l) := \lim_{D\to\infty} h^{(D)}(d,l) = 1 - e^{-\tau^*(b)F'(d)\sum_{v=l}^{V} g(v;d)} \tag{3.21}$$

where $\tau^*(b)$ is such that:

$$b = \lim_{D\to\infty} \mathbb{E}\left[\frac{S^{(D)}(D\tau)}{D}\right] = \int_0^1 \sum_{l=1}^{V} \Delta(x,l)dx \quad - \\ \int_0^1 \sum_{l=1}^{V} \Delta(x,l) e^{-\tau^*(b)F'(x)\sum_{v=l}^{V} g(v;x)} dx. \tag{3.22}$$

Proof. Refer to the Appendix. □

We next study a system of caches where probability requests for data objects, layers, and working-set size scales as a function of D and V.

As before, let F and G be two smooth, monotone increasing function with domain closed interval [0, 1], such that $F(0) = G(0) = 0$ and $F(1) = G(1) = 1$. We define the request probability for data object d and version v as D and V scales as follows:

$$q^{(D,V)}(d,v) = (F(d/D) - F((d-1)/D))\,(G(v/V) - G((v-1)/V)). \tag{3.23}$$

Based on the definition of F and G, we have
$\sum_{d=1}^{D}\sum_{l=1}^{V} q^{(D,V)}(d,v) = 1$ and $q^{(D,V)}(d,v) \geq 0$. Thus, $q^{(D,V)}(d,v)$ is a probability distribution determined by F and G. We use $\delta^{(D,V)}(d,l)$ to denote the size of layer l for data object d and $p^{(D,V)}(d,l) = \sum_{v=l}^{V} q^{(D,V)}(d,v)$ denotes the request probability for layer l of data object d. We define Riemann integrable Δ satisfying $\Delta(d/D, l/V) = \delta^{(D,V)}(d,l)$ for all D, V, d and l for the theorem below.

Theorem 3.2. *Consider the system of caches which scales as a function of D and V. For large D and V, the hit probability for data object d and layer l, $h^{(D,V)}(d,l)$, is approximated by*

$$h^{(D,V)}(d,l) = \left(1 - (1 - p^{(D,V)}(d,l))^{(t^*(B)-1)}\right) \tag{3.24}$$

where $t^*(B)$ is such that:

$$B = \sum_{d=1}^{D}\sum_{l=1}^{V} \delta^{(D,V)}(d,l)\left(1 - (1 - p^{(D,V)}(d,l))^{(t^*(B)-1)}\right). \tag{3.25}$$

In the limit $D \to \infty$ and $V \to \infty$, the hit probability for data object d and layer l is given by:

$$h(d,l) := \lim_{V\to\infty}\lim_{D\to\infty} h^{(D,V)}(d,l) = 1 - e^{-\tau^*(b)F'(d)G'(l)} \tag{3.26}$$

where $\tau^*(b)$ is such that:

$$b = \lim_{R\to\infty}\lim_{D\to\infty} \mathbb{E}\left[\frac{S^{(D,V)}(DV\tau)}{DV}\right] = \int_0^1\int_0^1 \Delta(x,y)dxdy \quad - \\ \int_0^1\int_0^1 \Delta(x,y)(e^{-\tau^*(b)F'(x)G'(y)})dxdy. \tag{3.27}$$

Proof. Similar to the proof of Theorem 3.1. □

3.2.10 Greedy hybrid LRU and LFU policies

In this subsection, we briefly describe two policies that adapt and store the best representation choice (MR or LR) for each cached data object's associated versions. These approaches are inspired by the work of [21] and aim to store objects as MR if there is a skewed popularity among its different versions. Conversely, if a data object is popular across multiple versions, it is stored as LR. Under our Greedy Hybrid LRU-type policy, when a request occurs for a version of a data object which is not present in the cache in any other version, the data object is fetched in the requested version and stored in its MR representation. If there is a request for a version of a data object which is different from a version that has already been cached, then both versions are stored as LR including all layers up to the maximum version requested. This policy is called "HLRU" in our numerical evaluation section.

Similarly we can define a static Greedy Hybrid LFU - type approach where the policy to decide which objects to include in the cache proceeds as follows. Data object versions are ranked in descending order of popularity none of which are initially cached. Take the most popular uncached object/version (d, v): If no cached version of d exists and there is room, then (d, v) enters the cache in its MR representation. If another version of d is in the cache and if there is room, then (d, v) enters the cache as LR the other cached version of d converts to LR. This process proceeds until the cache is full. This static Greedy Hybrid LFU policy is such that objects cached only in one version are stored as MR, otherwise LR. This policy is called "HLFU" in our numerical

evaluation section. The described policies are greedy under the constraint that for all data object d and version v, $s_{\mathrm{MR}}(d,v) \leq s_{\mathrm{LR}}(d,v)$ and, for all versions $v > 1$, $\min_{1\leq u<v\leq V} s_{\mathrm{MR}}(d,u) + s_{\mathrm{MR}}(d,v) \geq s_{\mathrm{LR}}(d,v)$.

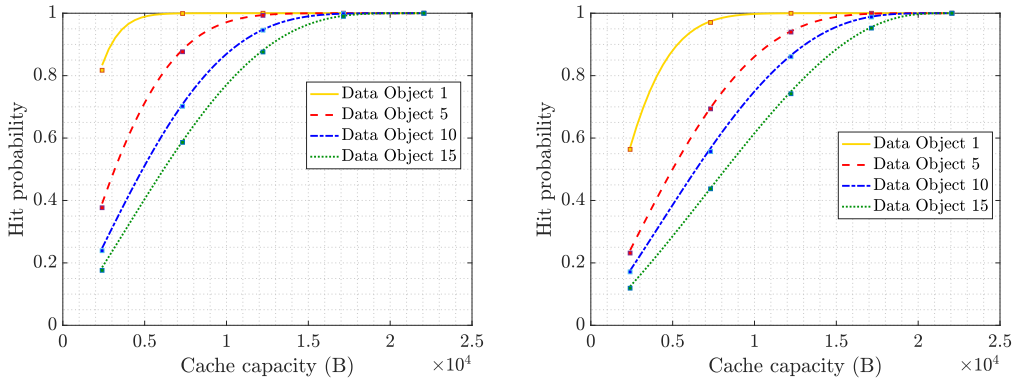
3.3 Numerical evaluation and simulation results

In this section, we perform extensive numerical evaluations based on the working set approximation and simulations of layered caching policies. The aim is to characterize the fundamental tradeoffs underlying the caching of data objects with LR and/or MR representations.

3.3.1 How accurate is the working set approximation for LLRU?

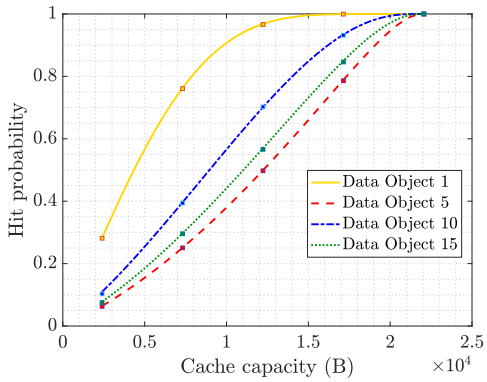
Setting. We consider a caching system with $D = 100$ data objects, each having $V = 4$ layered versions. The request probability $q(d)$ follows a Zipf distribution with parameter 0.8, while the request probability $q(d,v)$ for version v is uniformly selected from $(0, q(d))$, ensuring $\sum_{v=1}^{V} q(d,v) = q(d)$. Additionally, we impose $q(d,v_1) > q(d,v_2)$ for $v_1 < v_2$, reflecting the higher request frequency of lower versions. Requests for object d and version v follow a Poisson process with rate $q(d,v)$. The size of each layer is uniformly chosen from $[1, 240]$, ensuring a total object size of 240.

Results discussion. We plot the hit probability of data objects ranked 1, 5, 10, and 15 in Fig. 3.2. The squares represent the results obtained from simulations of LLRU policy, conducted over sufficiently long runs to ensure high accuracy. The lines are derived from the working set approximation for

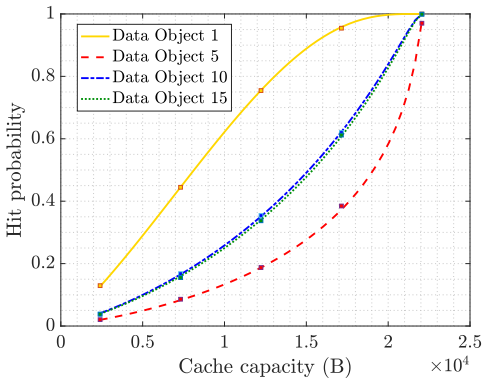


(a) Version 1

(b) Version 2



(c) Version 3



(d) Version 4

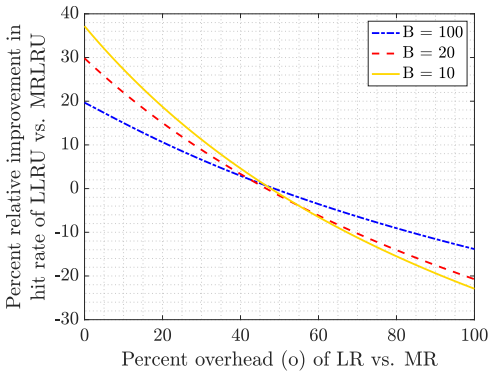
Figure 3.2: Hit probability against cache capacity for selected data objects under LLRU caching policy.

LLRU. The agreement between simulation results and approximation is nearly perfect for all practical purposes across all layers of a data object. We will use this approximation to address the questions posed at the beginning.

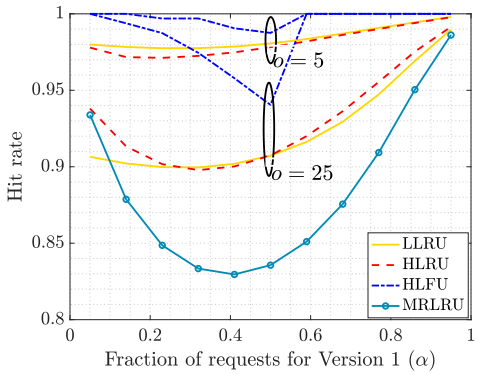
3.3.2 When are Multiple Representations (MR) better than Layered Representations (LR)?

Setting. We now examine a caching system with $D = 100$ data objects and $V = 2$ versions under multiple/layered representations, where request probability for a data object follows a Zipf distribution with parameter 0.8. Let $\alpha = q(d,1)/(q(d,1)+q(d,2))$ denote the request probability for Version 1 of MR/LR for data object d. Thus, the request probability for Version 2 of either MR or LR is $1-\alpha$. For the case of multiple representations, $\beta = s_{\text{MR}}(d,1)$ denotes the size of Version 1 and the size of Version 2 is 1, i.e., $s_{\text{MR}}(d,2) = 1$ for data object d. The size of Version 1 and 2 under layered representation is given by $s_{\text{LR}}(d,v) = (1+o)\cdot s_{\text{MR}}(d,v)$, where o is the percent overhead of LR vs. MR. As before the request arrival process is modeled as a Poisson process and we set the total request rate, λ, to $= 1$.

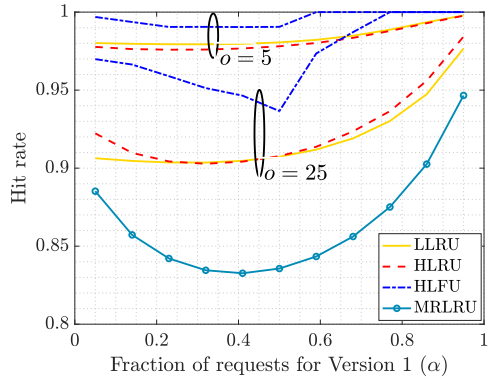
Results discussion. In Fig. 3.3a, we show the percentage relative improvement in the hit rate of LLRU (data objects are stored in LR) compared to MRLRU (data objects are stored in MR) for varying cache capacities (10, 20, and 100). As the cache capacity increases, the observed improvement decreases. This trend emerges because the hit rate for MRLRU and LLRU converges to 1 with increasing cache capacity, regardless of overhead. Ultimately,



(a) Comparison of hit rate of LLRU to MRLRU for $\alpha = 0.5$.



(b) Comparison of hit rate of different caching policies with $B = 100$.



(c) Comparison of hit rate of different caching policies when fraction of requests for versions of odd numbered data objects is $(\alpha, 1-\alpha)$ and $(0.5, 0.5)$ for the rest with $B = 100$.

Figure 3.3: Performance comparison of LR vs. MR for $\beta = 0.5$ as a function of percent overhead and fraction of requests.

a sufficiently large cache achieves the optimal hit rate of 1. Consequently, for such large cache capacities, there will be no difference in hit rates between LLRU and MRLRU, leading to no relative improvement.

Moreover, in Fig. 3.3b, we plot the hit rate under two scenarios: one where all data objects are exclusively stored in LR (with a cache utilizing LLRU) and another where they are stored in MR (with a cache utilizing MRLRU). This is presented as a function of fraction of requests for Version 1. We show the results for two different overhead values of 5 and 25. These overhead values represent the extremes for SVC vs. AVC overhead, see [16]. Additionally, we remind the reader of the HLRU and HLFU policies, see Section 3.2.10, that are capable of adapting the optimal representation for each data object. The rationale behind these approaches is to minimize the storage space occupied by the data object. Initially, the data object is stored in MR, given that $s_{\mathrm{MR}}(d,i) < s_{\mathrm{LR}}(d,i)$ for i equal to 1 or 2. However, if there is an additional request for the other version, the data object is then stored in LR, considering that $s_{\mathrm{LR}}(d,2) < s_{\mathrm{MR}}(d,1) + s_{\mathrm{MR}}(d,2)$.

We note that for $o = 25$, MRLRU performs better or comparable to LLRU when the fraction of requests for different versions is skewed, though this is not necessarily true for lower overhead values for example $o = 5$. Additionally, the hybrid variant of LRU, HLRU, designed to minimize the space occupied by each data object, consistently performs either as well as or better than LLRU for the presented overhead values. We perform a similar study for another scenario where we fix the fraction of requests for Version 1 of even-

numbered data objects at 0.5 and vary the fraction of requests for Version 1 of odd-numbered data objects using the parameter α, i.e., the fraction of request is α for Version 1 and $1 - \alpha$ for Version 2. The results are plotted in Figure 3.3c, and once again, the HLRU outperforms or matches LLRU. At last, we draw reader's attention to static Greedy Hybrid, HLFU, which uses the knowledge of popularities of data objects and versions to determine what to cache. This policy consistently outperforms all the other policies irrespective of the overhead values.

In summary, we note substantial performance benefits favoring layered representations over multiple representations, especially for reasonable percent overhead (o less than 25). However, for $o = 25$, MRLRU may outperform LLRU if the request distribution is skewed, highlighting the need for policies that can dynamically adapt and store the optimal representation for each data object. Next, we study the different layered caching policies.

3.3.3 Study of different layered caching policies

We begin with a study to compare the performance of different layered caching policies with two vs. only one version under layered representation for each data object. For this, we discretely vary the request rate for LR 1 for a fixed size of Version 1 and 2. Next, we study the performance comparison of LLRU policy for two vs. one version under layered representation as a function of request rate for LR 1 again for a fixed size of Version 1 and 2.

Setting. We have a caching system with $D = 100$ data objects. The

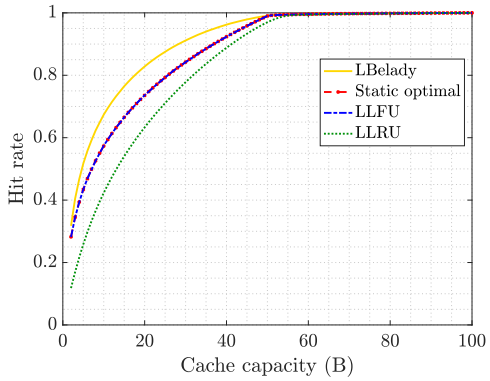
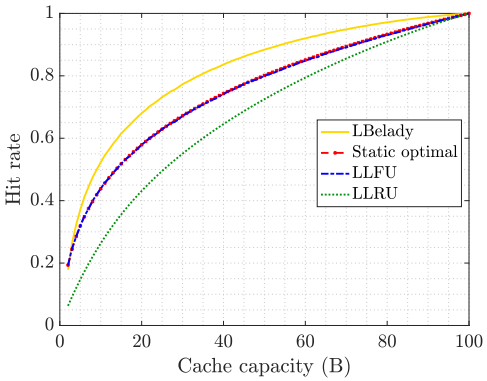


Figure 3.4: Hit rate against cache capacity under Layered caching policies for $(\alpha, \rho) = (0.99, 0.5)$.

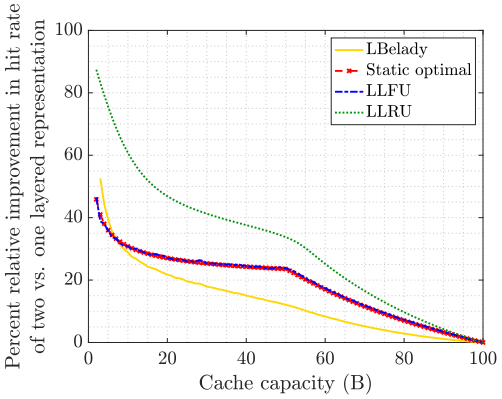
request probability for data objects follows a Zipf distribution with parameter 0.8. As before $\alpha = q(d,1)/(q(d,1) + q(d,2))$ denotes the fraction of requests for LR 1 of data object d when each data object has two versions. Thus, $1-\alpha$ is the fraction of requests for LR 2 or requests for both layers. For each data object, $\alpha = 0$ and $\alpha = 1$ correspond to all requests for both layers and only the first layer, respectively. Let $\rho = \delta(d,1)$ denote the size of Layer 1 for data object d, and the total size of each data object is 1, making the size of Layer 2 equal to $1-\rho$. The request arrival process follows a Poisson distribution with a total request rate equal to 1.

3.3.3.1 Layered caching policies

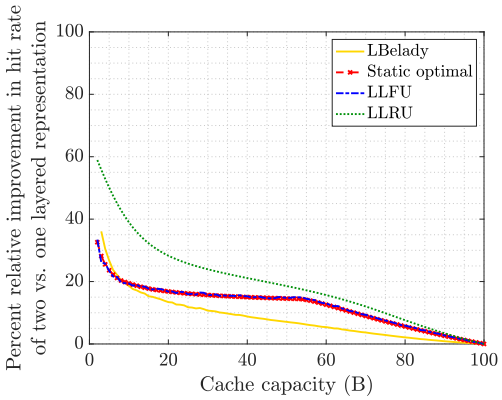
Results discussion. Fig. 4.4 depicts the hit rate for different layered caching policies and the optimal hit rate as a solution of the static optimal policy. We observe that the policy with knowledge of future arrivals, LBelady



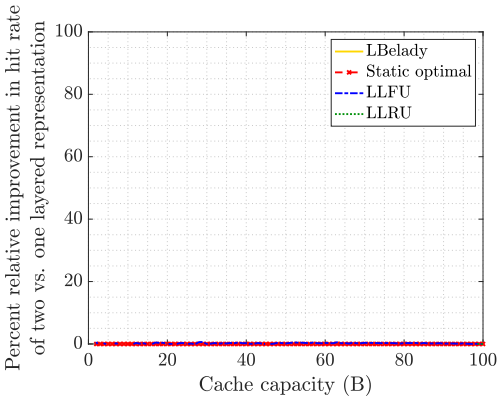
(a) Performance of different caching policies with 1 version under LR for each data object.



(b) Performance comparison for $(\alpha, \rho) = (0.99, 0.5)$.

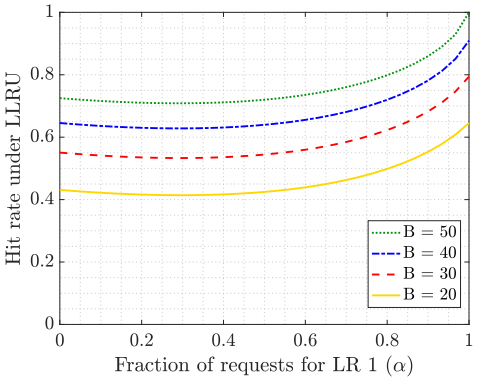


(c) Performance comparison for $(\alpha, \rho) = (0.9, 0.5)$.

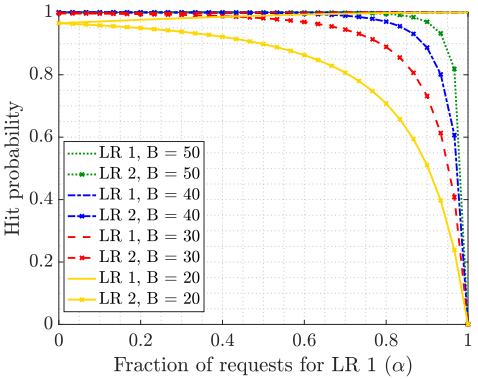


(d) Performance comparison for $(\alpha, \rho) = (0.5, 0.5)$.

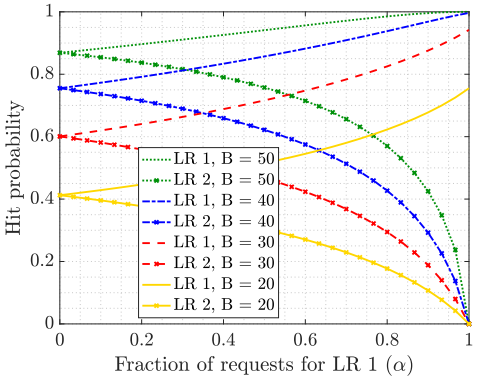
Figure 3.5: Performance comparison of two vs. one version under layered representation against cache capacity.



(a) Hit rate under LLRU with two versions under layered representation for each data object.



(b) Hit probability for Data Object 1.



(c) Hit probability for Data Object 10.

Figure 3.6: Performance of LLRU with two versions under layered representation for each data object against fraction of requests for LR 1 and $\rho = 0.5$.

performs the best, following that both LLFU and Static optimal have similar performance. Thus, LLFU, which keeps track of the number of arrivals for each data object and version, is the optimal policy among the class of policies without the knowledge of future arrivals. Finally, the LLRU policy, which does not require knowledge of the arrival process nor keep track of the number of arrivals for each data object and version has comparable performance.

3.3.3.2 Comparison of hit rate for two vs. one version under layered representation for different caching policies: discrete values for fraction of requests for version 1

Results discussion. As a baseline, we first show the hit rate under different layered caching policies when each data object consists of only 1 version in Fig. 3.5a. We then plot the percent relative improvement in hit rate of two vs. one version under layered representation with different layered caching policies for different values of α and ρ in Fig. 3.5. As α increases for fixed ρ, we observe an improvement in relative performance for all policies at a given cache capacity. Additionally, for cache capacity equal to 100 the percent relative improvement is 0 because all policies achieve the best possible hit rate.

3.3.3.3 Comparison of hit rate for two vs. one version under layered representation under LLRU: varying fraction of requests for Version 1

Results discussion. Fig. 3.6a shows the hit rate under the LLRU caching policy for different cache capacities as a function of the fraction of requests for LR 1, α. We observe a non-monotonic behavior for the hit rate

for all cache capacities. This is explained through Figures 3.6b and 3.6c, where we show the hit probability for both versions of data object 1 and 10, respectively, for different cache capacities. In both figures, the hit probability for LR 1 increases as the fraction of requests for Version 1 increases from left to right, while decreasing for LR 2. Since the hit rate is a convex combination of hit probabilities for LR 1 and LR 2, we observe the non-monotonic behavior in Fig. 3.6a. Thus, for a fixed size of Version 1, the performance is non-monotonic in the fraction of requests for LR 1.

3.3.4 Impact of layers' sizes and popularity on performance

In this section, we study the impact of layer size and popularity of layers on the hit rate. More specifically, we will fix the size of layers and offer guidance on how to set the popularity of versions and thus layers that is optimal. Similarly, for a fixed popularity of versions, we address the optimal setting of the size of each layer. We will do this first for the case where data objects have 2 versions and then 3 versions.

Setting. For this section, we have a caching system with $D = 100$ data objects. The request probability for data objects follows a Zipf distribution with parameter 0.8. For the case of 2 versions, we remind the reader about α, which denotes the fraction of requests for Version 1 of data object d and $\rho = \delta(d, 1)$ denotes the size of Layer 1 of data object d. With 3 versions for each data object, we let $\zeta = q(d, 1)/(q(d, 1) + q(d, 2) + q(d, 3))$, and $\eta = q(d, 2)/(q(d, 1) + q(d, 2) + q(d, 3))$ denote the fraction of requests for Version

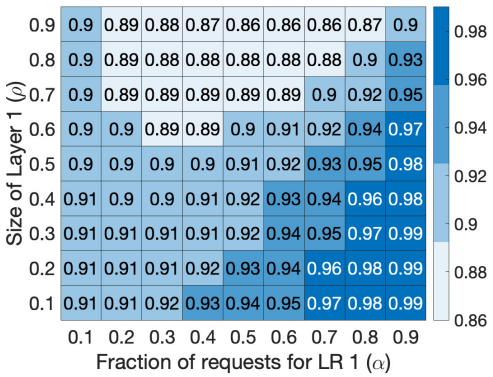
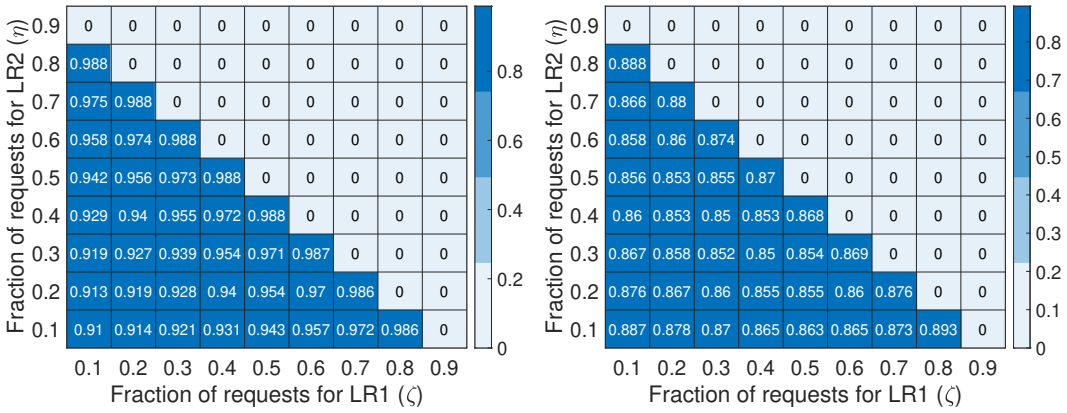


Figure 3.7: Performance of LLRU for different size and popularity for a cache capacity of 20.

1 and 2 respectively. Thus, the fraction of requests for Version 3 is $1 - \zeta - \eta$. We use $\rho = \delta(d, 1)$ and $\kappa = \delta(d, 2)$ to denote the size of Layer 1 and Layer 2, making the size of Layer 3 equal to $1 - \rho - \kappa$.

3.3.4.1 How to set the size and popularity when each data object has 2 version?

Results discussion. We show the performance of cache under LLRU caching policy in Fig. 3.7 for different values of fraction of requests for Version 1 and size of Layer 1. We observe that for a fixed fraction of requests for LR 1, as the size of Layer 1 decreases, the performance improvement is monotonically increasing. Also, as already observed in the previous section, the same is not true for the fixed size of Layer 1 and the increasing fraction of requests for LR 1. Furthermore, if both the size and fraction of requests vary simultaneously, possibly along a diagonal, the hit rate does not follow a monotonic pattern. The last observation is significant improvements occur



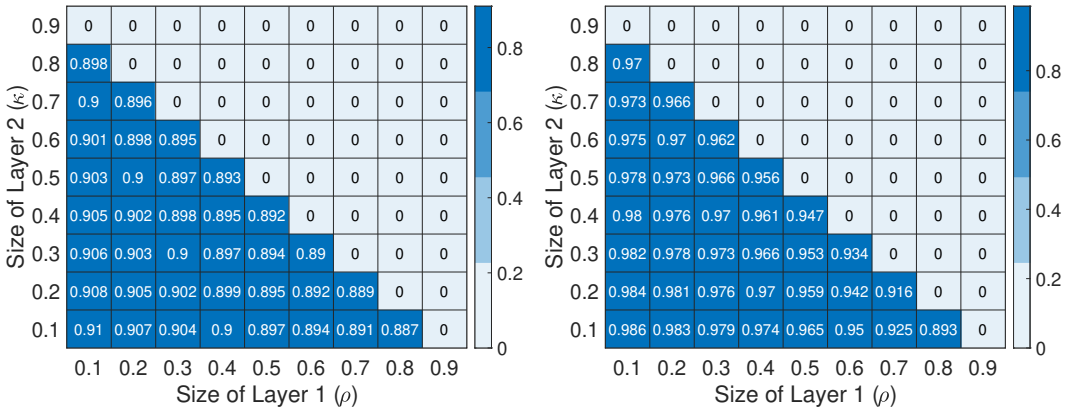
(a) Performance for $(\rho, \kappa) = (0.1, 0.1)$. (b) Performance for $(\rho, \kappa) = (0.8, 0.1)$.

Figure 3.8: Performance of LLRU for different popularities of versions when there 3 versions for $B = 80$.

with an increasing fraction of requests for LR 1 and a decreasing size of Layer 1.

3.3.4.2 How to set the size and popularity when each data object has 3 versions?

Results discussion. We show the performance of cache under LLRU caching policy in Fig. 3.8 for different values of fraction of requests for LR 1 and LR 2 under different fixed sizes of layers. We limit ourselves to a scenario where the fraction of requests for any version is at least 0.1 and thus for infeasible pairs of (ζ, η) we set the hit rate value to 0. In Fig. 3.8a, we observe that the maximum hit rate is observed for $(\zeta, \eta) = (0.8, 0.1)$, i.e., if most of the requests are for LR 1, which also has a small size, one observes the maximum hit rate. Similarly, in Fig. 3.8b we observe the maximum hit rate for $(\zeta, \eta) = (0.8, 0.1)$ even though the size of layer 1 is the maximum of all. This



(a) Performance for $(\zeta, \eta) = (0.1, 0.1)$. (b) Performance for $(\zeta, \eta) = (0.8, 0.1)$.

Figure 3.9: Performance of LLRU for different popularities of versions when there 3 versions for $B = 80$.

is a result of the condition that the presence of a higher layer implies all layers lower than that must also be present in the cache. In addition, we observe that for a fixed value of ζ and increasing η, the hit rate is not monotonic. Thus, a naive approach to selecting the popularity might not be optimal. We show similar results for the case when popularity is fixed, and we need to select the optimal size of layers in Fig. 3.9.

3.3.5 Is it beneficial to increase the number of versions for a data object?

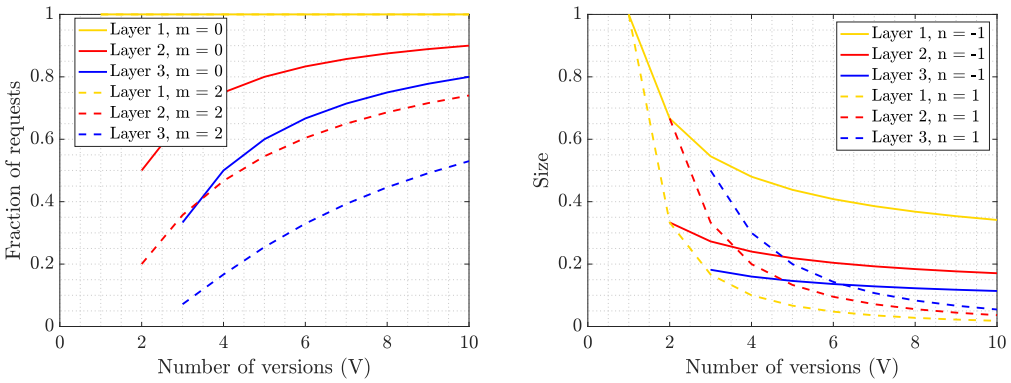
Setting. We manage 100 data objects, and the request probability for each data object follows a Zipf distribution with parameter 0.8. We scale the number of versions as V and correspondingly vary the request probability for data object d's vth version as $q^{(V)}(d, v) = \frac{(V-v+1)^m}{\sum_{i=1}^{V}(V-i+1)^m}$ while size varies as

$s^{(V)}_{\text{LR}}(d,v) = \sum_{l=1}^{v} \delta^{(V)}(d,l)$ where $\delta^{(V)}(d,l) = \frac{(l)^n}{\sum_{i=1}^{V}(i)^n}$. We plot the request probability, $p^{(V)}(d,l) = \sum_{v=l}^{V} q^{(V)}(d,v)$, for the first three layers and size, $\delta^{(V)}(d,l)$, as a function of number of versions in Fig. 3.10a and Fig. 3.10b, respectively.

Results discussion. We conduct a performance comparison of LLRU with V vs. 1 versions under layered representation in Fig. 3.10. In Figures 3.11a, 3.11b, 3.11c, and 3.11d we show the hit rate under LLRU against the number of versions for different values of m and n. We empirically observe that both the popularity and size of layers need to increase/decrease at a certain rate to see benefits in terms of the hit rate. In Fig. 3.11a, the hit rate is monotonically decreasing in the number of versions whereas by increasing the value of m for same n, we see a non-monotonic behavior, see Fig. 3.11b. This points to the subtle ways in which the overall hit rate depends on the number of versions, popularity, and size characterization. In addition, we observed that the hit rate is monotonic in the number of versions for all values of $m > 0$ and $n \geq 0$.

3.4 Conclusion

The efficient management of the large amounts of data required by emerging delay-constrained applications, e.g., multiplayer VR gaming and NN-based inference, will require judicious use of caching which exploit, when appropriate, hierarchies of data object representations that enable tradeoffs between data object's size and quality. To address this, in this chapter, we

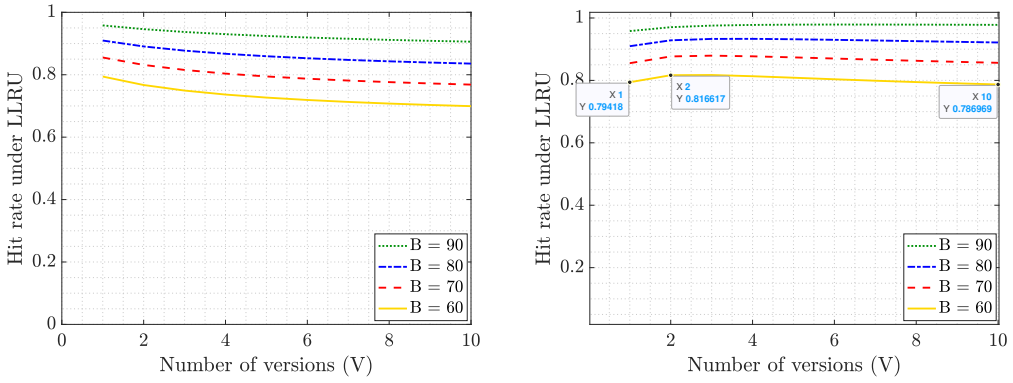


(a) Fraction of requests for layers.

(b) Size of layers.

Figure 3.10: Popularity and size characterization for different values of m and n as a function of number of versions.

have studied caching policies optimized for data objects with multiple versions and layered representations. Based on numerical analysis and simulation, the benefits of LR are substantial even if in some settings such hierarchical representations incur additional overheads. To make the most of such representations it is critical to understand the impact that the incremental size of layers and the level of demand for different versions will play. This chapter explores these impacts and suggests when, for example, additional layers may be of value, and when they may in be counterproductive, towards enhancing performance.



(a) m = 0, n = -1 (b) m = 2, n = -1

(c) m = 0, n = 1 (d) m = 2, n = 1

Figure 3.11: Comparison of LLRU with V vs. 1 version under layered representation for request probability of version v of data object d given by $q^{(V)}(d,v) = \frac{(V-v+1)^m}{\sum_{i=1}^{V}(V-i+1)^m}$ and size of lth layer by $\delta^{(V)}(d,l) = \frac{(l)^n}{\sum_{i=1}^{V}(i)^n}$.

Chapter 4

Inferring Causal Relationships to Improve Caching for Clients with Correlated Requests: Applications to VR

This chapter[1] tackles edge-cache design when client requests are correlated—as they are, for example, when groups of users explore the same VR scene and therefore ask for the same objects in succession. We introduce a grouped-client request model that extends the classic Independent Reference Model to capture such correlations and use it to analyse mainstream policies. The analysis shows an intriguing split: with small to medium caches, the frequency-based LFU rule is optimal, but once the cache is large enough, the recency-based LRU rule becomes superior. We then introduce LFRU: a lightweight online caching policy that adapts to structured correlations when present, outperforming both LRU and LFU across cache sizes.

[1]This chapter is based on the work submitted to Infocom 2025:

- A. Bari, G. De Veciana, Y.Zhou, "Inferring Causal Relationships to Improve Caching for Clients with Correlated Requests: Applications to VR," *Submitted to Infocom 2026*

Agrim Bari led the formulation of the problem, the design of policies, execution of experiments, and the writing of the paper.

4.1 Introduction

Managing shared edge caching. Efficient management of shared edge caches plays a crucial role in modern networked systems, reducing latency, alleviating backhaul network congestion, and improving client experience. However, ensuring data availability for multiple clients on the limited shared edge caches with diverse and dynamic request patterns presents a significant challenge. To maximize the overall performance, there are two decisions to be made, first, about what data to place in the cache (placement), second, what data to evict when storage constraints are reached (eviction).

Existing heuristic caching policies. Traditional caching policies rely on well-established heuristics to determine cache placement and eviction. Least Recently Used (LRU) prioritizes the eviction of objects accessed least recently, making it effective when request patterns exhibit strong temporal locality. Least Frequently Used (LFU) retains the most commonly requested objects over time, excelling in settings where demand popularity remains relatively stable. However, in many scenarios, client data requests exhibit correlations which can be dynamically changing.

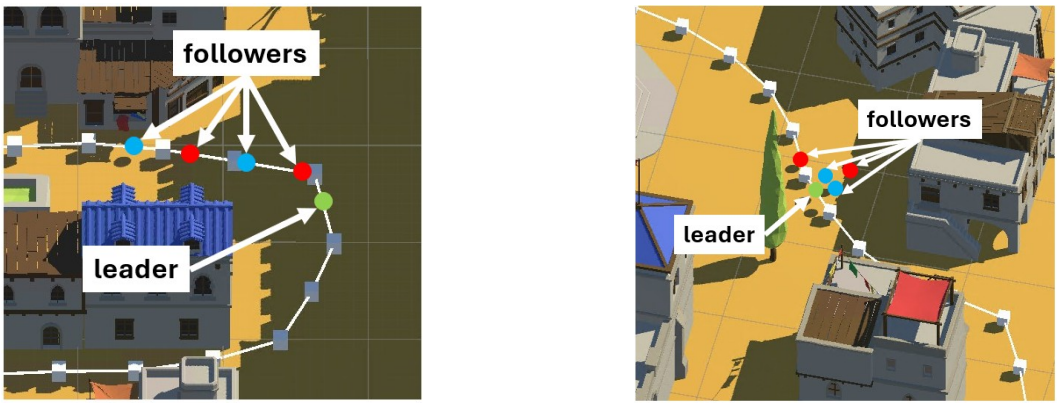
Correlation in client requests and applications. In many real-world applications, the patterns of request of client data are not independent but exhibit strong correlations due to shared contexts, coordinated activities, or inherent behavior of the system. For example, in collaborative Virtual Reality (VR) environments, correlations in client requests can arise explicitly, such as when students follow their teacher through a VR space and request the same

data objects with some delay. Alternatively, correlations may emerge implicitly when multiple clients independently visit a popular virtual location, such as New York, leading to overlapping content requests. Similarly, in collaborative editing platforms such as Google Docs or GitHub, employees working on shared documents or repositories often access overlapping data files in rapid succession.

A similar structure exists in edge computing, where client-generated tasks rely on microservices that must be loaded into the edge server's memory before execution. In many applications, clients invoke specific sequences of microservices due to inherent task dependencies. For instance, a request for an authentication service is frequently followed by requests for data processing or storage services, with multiple clients exhibiting similar access patterns. These correlations, whether arising from user behavior, system architecture, or application workflows, present an opportunity to optimize caching strategies, improving efficiency and reducing service latency.

Different client request patterns in VR.

In VR environments, client request patterns can exhibit different forms of correlation, as illustrated in Fig. 4.1. One common scenario, which we denote as *structured following*, occurs when followers replicate their leader's requests sequentially. For example, in a virtual museum setting, a teacher guiding students through exhibits results in each student accessing the same content in the same order as the teacher. In contrast, *unstructured following* arises when followers are distributed around the leader, leading to a staggered



(a) Structured following.

(b) Unstructured following.

Figure 4.1: Different types of correlations in a VR environment, example relative positions of clients.

and less predictable request pattern, such as when students explore a VR environment independently, but still request content influenced by the teacher's interactions.

In this work, we introduce a unified model denoted *grouped client request model* to capture structured and unstructured following behaviors. This model extends the traditional Independent Reference Model (IRM) by explicitly accounting for correlations among client requests. Furthermore, we propose a caching policy that dynamically infers causal relationships between client requests. In scenarios characterized by structured following, our approach significantly outperforms the conventional LRU policy and others by prioritizing the retention of objects likely to be requested by (follower) clients based on these inferred dependencies.

4.1.1 Related work

Analytical works on caching policies and approximation. We restrict our review to the most relevant papers in our work. [11, 29] summarize significant early work in the design of caching policies, and [22] describes analytical methods and evaluation results for the performance assessment of caching strategies. The aim of any caching policy is to achieve efficient cache utilization. This efficiency is measured primarily by the cache hit rate, which is the averaged fraction of data object requests for which the data object is in the cache when requested.

Besides hit rate, other design objectives for caching policies are ease of implementation, low operational overhead, and adaptability to fluctuations in access/request patterns. An important difference among caching policies is in what they evict when the cache is full. Under Least Recently Used (LRU), the cache is consistently updated to hold the most recently requested data objects, enabling it to leverage the temporal locality of data object requests. Notably for LRU under the Independent Reference Model (IRM), where each data object is requested independently of any past requests, the invariant distribution assuming data objects of the same size [38] and an approximation for the hit rate [12, 15, 9, 20] have been obtained. In particular, [12] describes the working-set approximation for hitting probabilities, the fraction of requests for a data object for which the object is in the cache. This approximation has been shown to be accurate as the number of objects scales [15, 20]. In this chapter, we define a new working set approximation and use that to find the

hit probability for requests from different clients under a grouped client request model defined later.

Under the IRM model, for a fixed cache capacity with same-size data objects, caching the most popular data objects is optimal for causal policies [2]. Least Frequently Used (LFU) performs optimally under stationary regimes of request patterns by replacing cached data objects based on the frequency measurements of past requests. An interesting work by [28] shows that a variant of LRU that infers the instantaneous request rate subject to the history of requests can come arbitrarily close to the optimal LFU algorithm. [30] shows that even for strongly correlated request patterns, LFU is still optimal among causal policies. However, while LFU may be effective in stationary scenarios where access patterns remain relatively constant, it may struggle to perform optimally in non-stationary regimes where the dynamics of data access change over time.

Machine Learning-Based Caching Approaches.

Recent advancements in Machine Learning (ML) have significantly influenced the development of intelligent and adaptive caching strategies. Approaches such as those in [49, 50, 17] leverage supervised learning to predict content popularity and optimize caching decisions. [49] focuses on long-term predictions, utilizing deep learning models to forecast future content demand. By contrast, [17] emphasizes the ability to adapt to short-term, immediate fluctuations in user demands, particularly in edge networks. While both methods demonstrate the potential of ML in improving caching performance, they

are heavily reliant on predictive models that require substantial amounts of training data. Moreover, they typically struggle to adapt to rapidly changing workloads, especially when faced with unpredictable shifts in user behavior. This limitation is mitigated by our proposed LFRU policy, which does not depend on predictive modeling. Instead, LFRU adapts in real-time based on observed correlations between client requests, making it more responsive to dynamic workloads without the need for extensive training data.

The authors of [39] and [42] employ reinforcement learning (RL) to make cache eviction decisions based on factors such as access patterns, object size, and content popularity. These RL-based approaches dynamically adjust their caching decisions in response to changing network conditions, thereby optimizing content delivery efficiency. While RL-based methods are effective in handling dynamic environments, they typically require continuous retraining to maintain accuracy as access patterns evolve. This presents a challenge in rapidly changing contexts, such as VR applications, where request patterns can fluctuate significantly. LFRU, however, does not rely on retraining or predictive modeling. Instead, it infers causal relationships between client requests in real-time, ensuring efficient cache eviction decisions even as access patterns change unpredictably.

The works most closely aligned with our approach are [26] and [18]. In [26], the authors use a neural network to model the inter-relationships between content requests. By learning patterns of dependencies among requests, [26] aims to optimize cache eviction decisions by prioritizing content that is

more likely to be requested soon. In contrast, LFRU builds similar relationships across client requests but does so in a computationally more efficient manner than neural networks. This makes LFRU a more lightweight solution, especially in resource-constrained environments.

In [18], cache decisions are made based on the Follow-The-Regularized-Leader algorithm, which optimizes the selection of actions by combining historical data with regularization penalties. Similarly, LFRU makes cache eviction decisions based on the observed behavior of clients and infers which clients are more likely to be followed by others. While both methods aim to optimize decision-making based on past behavior, LFRU offers a more direct and computationally less expensive approach by focusing on observed client correlations rather than relying on complex models or algorithms.

Finally, [47] explores hybrid strategies that integrate machine-learned predictions with traditional caching techniques. This combination seeks to improve the competitive ratio of caching algorithms despite potential inaccuracies in predictions. While hybrid methods have shown promise in improving caching performance, they still rely on predictive models, which can incur higher computational costs and may struggle with rapidly changing workloads. LFRU, in contrast, offers a purely observational approach that does not rely on predictions or complex models, providing a more efficient alternative in environments where quick adaptation and low computational overhead are critical.

4.1.2 Contributions and organization

In summary, we make the following key contributions:

- We introduce the *grouped client request model*, a generalization of IRM that captures different types of correlations in client requests.
- We derive a *working-set approximation* for computing hit probabilities under LRU and show that LFU is suboptimal for large caches in correlated request settings.
- We propose *Least Following and Recently Used (LFRU)*, a lightweight online caching policy that adapts to structured correlations when present, outperforming both LRU and LFU across cache sizes.
- We develop VR-based datasets to capture different types of correlated client requests and empirically show that LFRU improves cache hit ratios by up to 2.9× over LRU and 1.9× over LFU.

The remainder of this chapter is organized as follows. In Section 4.2, we describe the system model and present the working-set approximation for LRU under the grouped client request model. This section also includes an approximation for calculating hit probabilities for different clients. In Section 4.3, we introduce our caching policy that leverages inferred causal relationships. We then empirically evaluate and compare different caching policies using a dataset emulating a VR environment in Section 4.4. Finally, we conclude the chapter in Section 4.5.

4.2 System Model, analysis and simulation results

4.2.1 Model for cache

We shall consider a simple cache with capacity b bytes. The cache stores various data objects to serve future requests from a client population. Due to practical and cost constraints, the cache capacity typically is not enough to store all data objects.

4.2.2 Model for grouped client request patterns

We consider a fixed set of data objects, denoted by $\mathcal{D}$, where D is the total number of data objects that a population of clients can request. Clients are represented by the set $\mathcal{C} = \{1, 2, \ldots, C\}$, where C is the total number of clients. Each client belongs to exactly one of several distinct and non-overlapping groups, denoted by $\mathcal{G} = \{1, 2, \ldots, G\}$, where G is the total number of groups.

For each group $g \in \mathcal{G}$, let $\mathcal{D}^g$ represent the set of data objects that can be requested by clients in the group, and $\mathcal{C}^g$ represent the set of clients in the group. Each group has a designated leader client, denoted by l^g, along with a number f^g of followers.

For a data object $d \in \mathcal{D}$, we let $\mathcal{G}^d \subseteq \mathcal{G}$ denote the subset of groups that may request the data object d. Note that while groups do not overlap in terms of clients, they may share common data objects.

Definition 4.1 (Grouped Client Request Model)**.** *Each group's leader generates requests data objects independently of other requests, following a sta-*

tionary Poisson process. Followers in each group make the same data object requests but with a possibly random delay relative to the time the leader made the request.

Formally, let $\lambda^g(d)$ *denote the arrival rate of requests for data object* $d \in \mathcal{D}^g$ *by the leader* l^g*. The total arrival rate of requests generated by leader* l^g *is denoted as* $\lambda^g = \sum_{d \in \mathcal{D}^g} \lambda^g(d)$*. The probability that the leader requests data object* $d \in \mathcal{D}^g$ *is denoted as* $p^g(d) = \frac{\lambda^g(d)}{\lambda^g}$*.*

We define Δ_i^g *as a random variable representing the delay (or possible advancement) between the request of the* i*-th follower and the leader of group* g *for the same data object. The joint distribution of these delays across all followers in group* g *is* $(\Delta_i^g : i = 1, \ldots, f^g)$*. These delays can have an arbitrary joint distribution, independent but not identically distributed, or independent and identically distributed (i.i.d). The grouped sequence of requests from group* g *is modeled as a Marked Poisson Point Process (MPPP):*

$$\boldsymbol{A}^g = \left((A_n^g, D_n^g, (\Delta_{i,n}^g : i = 1, \ldots, f^g)) : n \in \mathbb{Z}^+\right) \sim MPPP(\lambda^g),$$

where A_n^g *is the arrival time of the* n*-th request from the leader of group* g*,* D_n^g *is a random variable representing the data object requested by the leader,* $(\Delta_{i,n}^g : i = 1, \ldots, f^g) \sim (\Delta_i^g : i = 1, \ldots, f^g)$ *represents the joint distribution of delays for the* n*-th request from the followers of group* g*, relative to when the leader made the request. These delays associated with followers requests are independent across different request instances* n*.*

The overall sequence of arrivals across all groups is denoted as $\boldsymbol{A} =$

$(\boldsymbol{A}^g : g \in \mathcal{G})$. *The processes* $\boldsymbol{A}^g$ *for different groups* $g \in \mathcal{G}$ *are independent MPPPs. The probability that the leader of group* g *requests data object* $d \in \mathcal{D}^g$ *is* $\mathbb{P}(D_n^g = d) = p^g(d)$.

Finally, we let $s(d)$ *denote the size of data object* d*, and let* $\boldsymbol{\Lambda} = (\lambda^g(d) : g \in \mathcal{G}, d \in \mathcal{D}^g)$ *represent the vector of request rates for each group leader and data object.*

Remark 4.1 (Generality of the Grouped Client Request Model)**.** *The proposed model is flexible and can represent various scenarios where client requests are grouped/correlated. For example:*

1. *Independent Reference Model (IRM): If* $f^g = 0$ *for all* $g \in \mathcal{G}$*, the model reduces to the Independent Reference Model, where client requests are independent and uncorrelated.*

2. *Structured follower requests: If the delays* Δ_i^g *are fixed and structured, such as* $\Delta_i^g = \delta_i + \Delta_{i-1}^g$ *for* $i > 1$ *where* δ_i *are positive constants, then each leader's request is followed by a predictable sequence of follower requests. This models scenarios like a teacher guiding students in a VR environment, where both the teacher and students request data objects that fall within their field of view. Here, the students follow the teacher in a fixed sequence, making requests one after another.*

3. *Unstructured random follower requests: If the delays follow a random distribution, such as* $\Delta_i^g \sim U[\alpha_i, \beta_i]$*, where* α_i *can be negative, the model*

represents a scenario where follower requests are randomly distributed around the leader's request. This captures a more dynamic VR setting, where students follow a teacher but do not adhere to a strict structure. Students may request data before or after the teacher.

These examples demonstrate the model's ability to handle both deterministic and random correlated patterns for client group requests.

Remark 4.2 (Extending the Grouped Client Request Model)**.** *While this model assumes that followers always request data after their leader, it can be easily extended to allow followers to randomly opt out of following the leaders requests. This is a potential direction for our future work.*

4.2.3 Hit probabilities for leaders and followers under LRU under the grouped client request model

The Least Recently Used (LRU) policy evicts the least recently accessed data object in the cache when a new object is requested, provided that the new object is not already in the cache and there is no space available to cache. Suppose d is requested at time 0. Under LRU, assuming the cache orders data objects from most recently used to least recently used, d initially occupies the bottom position. Over time, after some data objects (other than d) are requested, d moves to the top of the cache and is subsequently evicted, provided that it is not requested again before this happens. Now, we let $T_b(d)$ be a random variable representing the amount of time it would take to accumulate other unique data objects requests excluding d so as to fill the

cache. This variable, $T_b(d)$, is referred to as *characteristic time* of the data object $d \in \mathcal{D}$.

Quantifying $T_b(d)$ is important for determining the probability that d will still be in the cache under the LRU policy. In the literature, two approximations simplify this calculation, and these approximations become more accurate as the total number of data objects, D, increases (i.e., when $D \gg 1$). For further details, see [9, 20].

Approximation 1: For $D \gg 1$, the characteristic time $T_b(d)$ concentrates around its mean value. Therefore, $T_b(d)$ can be approximated by a deterministic value, $t_b(d)$, for each data object d.

Approximation 2: The dependence of $t_b(d)$ on the specific data object d can be neglected. This approximation is commonly used and justified in the literature [9, 20], particularly when the request probability $\sum_{g \in \mathcal{G}^d} p^g(d)$ is small relative to that of the remaining data objects request probabilities. This approximation becomes exact when the request probabilities are uniform.

Thus we introduce t_b^* as the mean characteristic time for any data object. Given the above approximations, t_b^* satisfies the following equation:

$$b = \sum_{d \in \mathcal{D}} \mathbb{P}\left(\text{data object } d \text{ was requested at least once in } [-t_b^*, 0]\right) s(d). \quad (4.1)$$

where the right hand side captures the size of set of data objects present in the cache at time 0 without loss of generality and since under LRU a data object stays in the cache for t_b^* amount of time after it is requested, we focus on the case whether a data object was requested at least once in the time

interval $[-t_b^*, 0]$. This is referred to as the working set approximation, which has been shown to be accurate as the number of data objects scales. We define an event $V_\tau^{l^g,i}$ as follows: the leader client l^g of group g makes a request at time τ, and $i-$th follower's request for the same data object does not occur within $[-t_b^*, 0]$. This condition is equivalent to $\{\tau + \Delta_i^g \notin [-t_b^*, 0]\}$. Let $p(d, t_b^*)$ denote the probability that data object d is requested at least once in $[-t_b^*, 0]$. We compute it in the following lemma.

Lemma 4.2 (Working Set Approximation)**.** *Consider a cache of size b that follows the Least Recently Used (LRU) eviction policy and serves a grouped client request pattern (see Section 4.2.2 for details). The probability that a data object d is requested at least once within the time interval $[-t_b^*, 0]$ is given by*

$$p(d, t_b^*) = 1 - e^{-\sum_{g \in \mathcal{G}^d} \int_{-\infty}^{\infty} \lambda_\tau^g(d, t_b^*)\, d\tau}, \tag{4.2}$$

where $\lambda_\tau^g(d, t_b^*) = \lambda^g(d) q_\tau^g(t_b^*)$. The function $q_\tau^g(t_b^*)$ is defined as:

$$q_\tau^g(t_b^*) = \begin{cases} 1 - \mathbb{P}\left(\bigcap_{i=1}^{f^g} V_\tau^{l^g,i}\right) & \text{if } \tau \in [-\infty, -t_b^*) \cup (0, \infty), \\ 1 & \text{if } \tau \in [-t_b^*, 0]. \end{cases} \tag{4.3}$$

Under Approximations 1 and 2, the characteristic time t_b^* is determined by solving the fixed-point equation:

$$b = \sum_{d \in \mathcal{D}} p(d, t_b^*) s(d). \tag{4.4}$$

Proof. To derive $p(d, t_b^*)$, we first compute the probability that no client in a group $g \in \mathcal{G}^d$ requests data object d within the interval $[-t_b^*, 0]$.

The leader of group g generates requests for d as a PPP with rate $\lambda^g(d)$. Consider the leader of group g requests d at time τ, then its i-th follower requests it at time $\tau + \Delta_i^g$. A request for data object d from a client in group g can fall in the interval $[-t_b^*, 0]$ either due to the leader's request occurring within the interval or due to a follower's request. Formally, we define $q_\tau^g(t_b^*)$ as the probability that at least one client in group g requests d in $[-t_b^*, 0]$, given that the leader made a request at time τ. This probability is given by

$$q_\tau^g(t_b^*) = \begin{cases} 1 - \mathbb{P}\left(\bigcap_{i=1}^{f^g} V_\tau^{l^g,i}\right) & \text{if } \tau \in [-\infty, -t_b^*) \cup (0, \infty), \\ 1 & \text{if } \tau \in [-t_b^*, 0]. \end{cases} \tag{4.5}$$

Since the leader requests arrive according to a PPP, we can define an inhomogeneous thinned PPP that models at least one client in group g requesting d within $[-t_b^*, 0]$, with rate $\lambda_\tau^g(d, t_b^*) = \lambda^g(d) q_\tau^g(t_b^*)$. We can now compute the probability that no client in group g requests d in $[-t_b^*, 0]$ as:

$$e^{-\int_{-\infty}^{\infty} \lambda_\tau^g(d, t_b^*)\, d\tau}. \tag{4.6}$$

Since the requests from different groups are independent, the probability that d is requested at least once in $[-t_b^*, 0]$ is

$$p(d, t_b^*) = 1 - \mathbb{P}(\text{No client in } \mathcal{G}^d \text{ requests } d \text{ in } [-t_b^*, 0]), \tag{4.7}$$

$$= 1 - e^{-\sum_{g \in \mathcal{G}^d} \int_{-\infty}^{\infty} \lambda_\tau^g(d, t_b^*)\, d\tau}. \tag{4.8}$$

Finally, under the LRU policy, the total size of cached data objects at time 0 is $\sum_{d \in \mathcal{D}} p(d, t_b^*) s(d)$. Under approximations 1 and 2, we can now find the

characteristic time t_b^* which satisfies

$$b = \sum_{d \in \mathcal{D}} p(d, t_b^*) s(d). \tag{4.9}$$

□

We now use this lemma to compute the hit probability for a data object requested by a client in a group. Let $p^L(d, g)$ denote the hit probability for data object d when requested by the leader l^g of group g. Similarly, let $p^F(d, g; i)$ represent the hit probability for a request from the i-th follower in group g.

For two clients $c_1, c_2 \in \mathcal{C}^g$ in group g, we define an event E^{c_1,c_2} as follows: client c_1 makes a request for a data object at time 0, and client c_2's request for the same data object does not occur within the time interval $[-t_b^*, 0]$. For example, if c_1 is the leader l^g and c_2 is the i-th follower in group g, then $E^{l^g,i}$ represents the event that the leader generates a request at time 0, but the i-th follower's request does not fall within $[-t_b^*, 0]$. This condition is equivalent to $\{\Delta_i^g \notin [-t_b^*, 0]\}$.

Theorem 4.3 (Hit Probability for Clients)**.** *Consider a cache of size b that follows the Least Recently Used (LRU) eviction policy and serves a grouped client request pattern (see Section 4.2.2 for details). Under Approximations 1 and 2, the hit probability for a data object d requested by leader l^g of group g is given by*

$$p^L(d, g) = 1 - [1 - p(d, t_b^*)] \left[\mathbb{P} \left(\bigcap_{i=1}^{f^g} E^{l^g,i} \right) \right]. \tag{4.10}$$

Similarly, under Approximations 1 and 2, the hit probability for a data object d requested by i-th follower of group g is

$$p^F(d,g;i) = 1 - [1 - p(d,t_b^*)] \left[\mathbb{P} \left(\left(\bigcap_{\substack{j=1 \\ j \neq i}}^{f^g} E^{i,j} \right) \cap E^{i,l^g} \right) \right]. \tag{4.11}$$

Proof. To calculate the hit probability for a data object d requested by the leader l^g of group g, assume without loss of generality that this request occurs at time 0. The leader's hit probability depends on two factors: first, the randomly distributed request from l^g's followers around time 0: l^g can get a hit if at least one follower in group g requests data object d within $[-t_b^*, 0]$ since the followers can request for a data object before their leader does (Section 4.2.2 for details), second, requests other than the randomly distributed request from l^g's followers around time 0: by Slivnyak's theorem [5], the remaining requests still follow a MPPP and thus l^g can get a hit if at least one client, from any group, requests data object d within the interval $[-t_b^*, 0]$ which is given by the earlier lemma under Approximations 1 and 2.

Using these two conditions, the leader's hit probability can be expressed as:

$$p^L(d,g) = 1 - [\mathbb{P}(\text{No follower in group } g \text{ requests } d \text{ in } [-t_b^*, 0])] \, [p(d,t_b^*)] \, . \tag{4.12}$$

From our earlier definition of event $E^{l^g,i}$, we know:

$$\mathbb{P}(\text{No follower in group } g \text{ requests } d \text{ in } [-t_b^*, 0]) = \mathbb{P} \left(\bigcap_{i=1}^{f^g} E^{l^g,i} \right). \tag{4.13}$$

Now, consider the hit probability for a data object d requested by the i-th follower of group g. Again, assume without loss of generality that this request occurs at time 0. As before the follower's hit probability depends on two considerations: first, requests from clients other than i-th follower in group g: i-th follower can get a hit if at least one other client in group g requests data object d within $[-t_b^*, 0]$, second, requests other than the requests already considered: again using Slivnyak's theorem [5], the remaining requests still follow a MPPP and thus i-th follower can get a hit if at least one client (from any group) requests the data object d within the interval $[-t_b^*, 0]$ which is given by the earlier lemma under Approximations 1 and 2.

Using these two conditions, we express the hit probability for follower i as:

$$p^F(d, g; i) = 1 - \mathbb{P}\left(\begin{smallmatrix}\text{No client except the } i\text{-th follower in}\\ g \text{ requests } d \text{ in } [-t_b^*,0]\end{smallmatrix}\right)[1 - p(d, t_b^*)] \tag{4.14}$$

From our earlier definition of events, we obtain:

$$\mathbb{P}\left(\begin{smallmatrix}\text{No client except the } i\text{-th follower in}\\ g \text{ requests } d \text{ in } [-t_b^*,0]\end{smallmatrix}\right) = \mathbb{P}\left(\left(\bigcap_{\substack{j=1\\ j\neq i}}^{f^g} E^{i,j}\right) \cap E^{i,l^g}\right). \tag{4.15}$$

□

In the coming section, we will use this theorem to understand the impact of client group structures.

Remark 4.3 (Calculations for different distribution(s) of Δ_i^g)**.** *We derive the following expressions for hit probabilities based on different distribution(s) of* Δ_i^g*:*

1. ***Structured follower requests:*** *If* $\Delta_i^g = i \cdot \delta$*, where* $\delta > 0$*, we can compute the concerned probabilities as follows:*

$$p(d, t_b^*) = 1 - e^{-\left(\sum_{g \in \mathcal{G}^d} \lambda^g(d) \left(t_b^* + f^g \cdot \min(\delta, t_b^*)\right)\right)}, \tag{4.16}$$

$$p^L(d, g) = p(d, t_b^*), \tag{4.17}$$

$$p^F(d, g; i) = \begin{cases} 1, & \text{if } \delta < t_b^*, \\ p(d, t_b^*), & \text{if } \delta \geq t_b^*. \end{cases} \tag{4.18}$$

4.2.4 Caching Policies

We will consider the set Π of stationary caching policies including both online and offline policies which possibly adapt the cached content based on incoming requests, and may have knowledge about future requests or request rates. For a given vector of request rates $\mathbf{\Lambda}$ and policy $\pi \in \Pi$, we define $\mathbf{h}_{\mathbf{\Lambda},\pi} = (h^{g,c}_{\mathbf{\Lambda},\pi}(d) : g \in \mathcal{G}, c \in \mathcal{C}^g, d \in \mathcal{D}^g)$, where $h^{g,c}_{\mathbf{\Lambda},\pi}(d)$ denotes the long-term fraction of requests for data object d from client c of group g that result in a cache hit.

4.2.5 Performance Metric

We evaluate the performance of a caching policy based on the overall cache hit ratio. For a given request rate vector $\mathbf{\Lambda}$ and policy π, the cache hit

ratio is defined as:

$$H_{\mathbf{\Lambda},\pi} = \sum_{g\in\mathcal{G}} \sum_{d\in\mathcal{D}^g} \lambda^g(d) h^{g,l^g}_{\mathbf{\Lambda},\pi}(d) + \sum_{g\in\mathcal{G}} \sum_{i=1}^{f^g} \sum_{d\in\mathcal{D}^g} \lambda^g(d) h^{g,i}_{\mathbf{\Lambda},\pi}(d). \tag{4.19}$$

4.2.6 An optimal (offline) static caching policy

An optimal static caching policy is one that decides which fixed set of data objects to retain in the cache to maximize cache hit ratio given the request rate vector $\mathbf{\Lambda}$. Let $\mathbf{x} = (x(d) : d \in \mathcal{D})$, where $x(d)$ is a binary variable indicating whether data object d is cached. We formulate the following optimization problem to maximize the cache hit ratio:

$$\max_{\mathbf{x}} \quad \sum_{g\in\mathcal{G}} \sum_{d\in\mathcal{D}^g} \lambda^g(d) x(d)(1+f^g) \tag{4.20a}$$

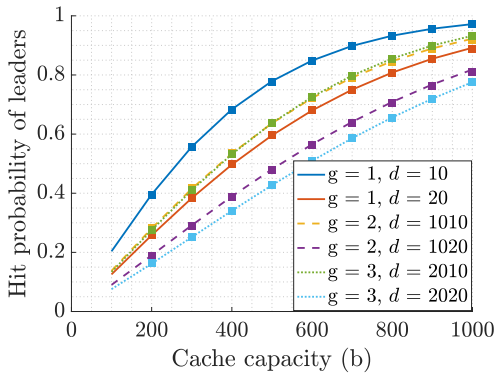
$$\text{s.t.} \quad \sum_{d\in\mathcal{D}} x(d)s(d) \leq b, \tag{4.20b}$$

$$x(d) \in \{0,1\}, \forall\ d \in \mathcal{D} \tag{4.20c}$$

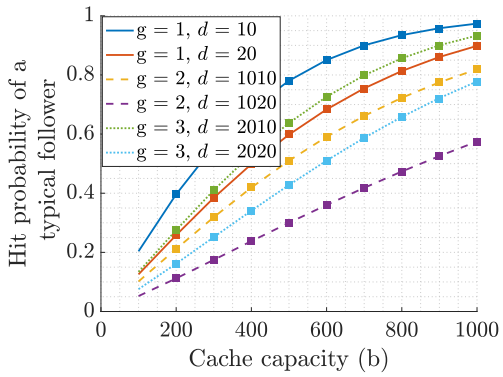
where constraint Eq. 4.20b ensures that the cache size does not exceed the capacity b.

4.2.7 Simulation results

In this subsection, we show the accuracy of the working set approximation for grouped client request pattern under the LRU caching policy and perform a comparative evaluation of different caching policies.



(a) Hit probability of leaders.



(b) Hit probability of a typical follower of each group.

Figure 4.2: Hit probability against cache capacity for different clients and data objects under LRU.

4.2.7.1 How accurate is the working set approximation for grouped client request pattern under LRU?

Setup. We consider a caching system with three groups ($G = 3$), where each group can request data objects from a distinct set. The data object sets for the groups are defined as follows:

$$\mathcal{D}^1 = \{1, 2, \ldots, 1000\}, \quad \mathcal{D}^2 = \{1001, 1002, \ldots, 2000\},$$
$$\mathcal{D}^3 = \{2001, 2002, \ldots, 3000\}.$$

Each group consists of one leader and several followers. The number of followers in each group is:

$$f^1 = 6, \quad f^2 = 4, \quad f^3 = 3.$$

The delay between the i-th follower (for all $i \in \{1, \cdots, f^g\}$) and the leader of group g is modeled as an independent and identically distributed (i.i.d.)

random samples from a uniform distribution:

$$\text{Group 1: } \Delta_i^1 \sim \text{U}[-10, 20], \quad \text{Group 2: } \Delta_i^2 \sim \text{U}[15, 30],$$
$$\text{Group 3: } \Delta_i^3 \sim \text{U}[-5, 40].$$

The request probability $p^g(d)$ for an object d in group g follows a Zipf distribution with parameter 1, identical across all groups. Requests from group leaders arrive according to a Poisson process, with the following rates:

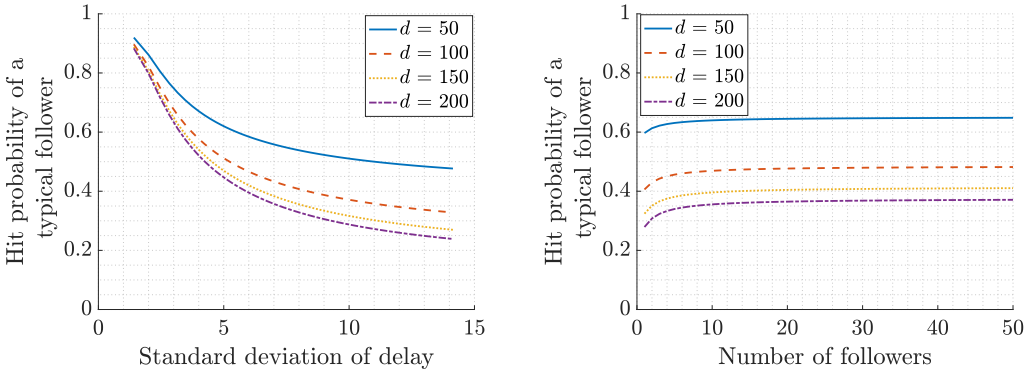
$$\lambda^1 = 10, \quad \lambda^2 = 8, \quad \lambda^3 = 12.$$

The size of objects is defined so that even-numbered objects have a size of 2, while odd-numbered objects have a size of 5 for all groups.

Results Discussion. Fig. 4.2 shows the probability of hit for various objects requested by clients (leaders and followers) in different groups. Since follower delays in each group are i.i.d., the hit probability for all followers in a group is identical. The squares in the figure represent simulation results for the LRU policy, obtained from long simulation runs to ensure reliable accuracy. The lines are derived from the working set approximation for LRU. As shown, the approximation aligns almost perfectly with the simulation results across all clients, demonstrating its high accuracy for practical applications.

4.2.7.2 Impact of client group structure on hit probability.

In this subsection, we use the working set approximation to analyze how the temporal overlap in follower requests and the number of followers affect



(a) Impact of standard deviation of delay.

(b) Impact of number of followers.

Figure 4.3: Impact of client group structure on hit probability for follower requests.

the hit probability for follower requests. We focus on the following simulation setup:

Setup. We consider a caching system with a single group of clients requesting objects from the set $\mathcal{D}^1 = \{1, 2, \ldots, 5000\}$. The group has f^1 followers, and the delay between the i-th follower and the group leader is modeled as a uniform random variable $\Delta_i^1 \sim \mathrm{U}[\alpha, \beta]$. The probability of a request $p^1(d)$ for a data object d follows a Zipf distribution with parameter 1. The group leader generates requests according to a Poisson process with rate $\lambda^1 = 20$. Object sizes are defined such that even-numbered objects have a size of 2, while odd-numbered objects have a size of 5.

Impact of standard deviation of delay. We fix the mean delay for follower requests, i.e., $0.5\ (\alpha + \beta) = 30$ and vary the standard deviation of the delay, $(\beta - \alpha)/\sqrt{12}$. The number of followers is set to $f^1 = 4$. Fig. 4.3a

shows the hit probability of a typical follower for different data objects as a function of standard deviation of delay. Since follower delays are independent and identically distributed, all followers have the same hit probability. As the delay variance increases (reducing the temporal locality in follower requests), the hit probability for each data object decreases. This demonstrates how temporal locality can be quantitatively linked to cache performance under the LRU policy.

Impact of number of followers. Here, we fix $\alpha = 0$ and $\beta = 60$, and vary the number of followers. Fig. 4.3b depicts the hit probability for different data objects as a function of the number of followers. As the number of followers increases, the likelihood of requests for the same data object with short delays between them increases. Consequently, the hit probability for each data object increases.

4.3 Caching based on inferred causal relations

In the previous section, we analyzed the performance of the LRU under the grouped client request model. As we will observe in Section 4.4, LRU performs suboptimally for small cache capacities compared to alternative caching strategies. To address this limitation, we propose a caching policy that infers temporal causal relationships between client requests and prioritizes eviction based on two key factors: (i) the frequency with which a client's request is followed by requests from other clients and (ii) the recency of requests. By leveraging these inferred dependencies, the proposed policy improves cache

performance across different cache capacities.

Our focus is on structured following, such as the interactions between teachers (leaders) and students (followers) navigating a VR environment. However, the grouping of clients is not known a priori and must be dynamically inferred and tracked by our caching policy. In the following section, we introduce the necessary notation and framework to formalize this approach.

4.3.1 Notation

Recall that the set of clients is denoted by $\mathcal{C} = \{1, 2, \ldots, C\}$, where $C = |\mathcal{C}|$ represents the total number of clients. Let $a = ((a_m, d_m, c_m) : m \in \mathbb{Z}^+)$ represent a sequence of requests ordered in time, where a_m is the arrival time of the m-th request, d_m is the data object requested at that time, and c_m is the client who made the m-th request. We denote the sequence of arrival requests generated by client c as $a^c = ((a_n^c, d_n^c) : n \in \mathbb{Z}^+)$, where a_n^c is the arrival time of the n-th request made by client c, and d_n^c is the data object requested at that time. We let $M(t)$ denote the set of data objects present in the cache memory at time t.

To identify the last client who requested a specific data object d before time t, we define the function:

$$c(d,t) = \begin{cases} \text{argmax}_c \left\{ \begin{matrix} a_n \cdot \mathbf{1}(d_n = d) \cdot \mathbf{1}(a_n < t) \\ : c \in \mathcal{C}, n \in \mathbb{Z}^+ \end{matrix} \right\} & \text{if client exists,} \\ -1 & \text{otherwise.} \end{cases} \tag{4.21}$$

where $\mathbf{1}(\cdot)$ is the indicator function. If no such client exists, the function returns -1.

Next, we define $n^c(t)$ as the index of the last request made by client c at or before time t:

$$n^c(t) = \underset{n}{\operatorname{argmax}} \left\{ a_n^c \leq t : n \in \mathbb{Z}^+ \right\}. \tag{4.22}$$

Definition 4.4 (Following event)**.** *We define a following event, where say client c_2 follows c_1 on a cache hit if: (1) c_2 requests a data object for which it experiences a cache hit and (2) c_1 is the client that most recently requested the same data object.*

Formally, suppose that client c_2's n-th request for data object $d_n^{c_2}$ at time $a_n^{c_2}$ results in a cache hit, i.e., $d_n^{c_2} \in M(a_n^{c_2})$, and that client c_1 was the last client to request $d_n^{c_2}$ before time $a_n^{c_2}$, i.e., $c(d_n^{c_2}, a_n^{c_2}) = c_1$. In this case, we say that client c_2 followed client c_1. We mathematically define this as follows:

$$f_n^{c_1,c_2} = \mathbf{1}(d_n^{c_2} \in M(a_n^{c_2}))\mathbf{1}(c(d_n^{c_2}, a_n^{c_2}) = c_1) \tag{4.23}$$

4.3.2 Caching policy

4.3.2.1 Least Following and Recently Used (LFRU (w))

The LFRU policy can improve on traditional LRU caching policy by considering inferred temporal relationships (following events) between clients' requests. This policy manages cache evictions by looking at both the recency of a request and how often other clients have followed the client who made the request. The goal is to keep data objects that were recently accessed and are

more likely to be accessed again soon, based on these past patterns of client requests.

To quantify these temporal relationships, we examine the last w requests made by each client, where w is a policy parameter. Specifically, we construct a matrix $F(t)$ at time t of size $C \times C$, where the entry in the c_1-th row and c_2-th column represents the number of times in the last w requests of client c_2, that client c_2 followed client c_1. More formally, we define the (c_1, c_2) element of the matrix as

$$F^{c_1,c_2}(t) = \begin{cases} \sum_{n=n^{c_2}(t)-w}^{n^{c_2}(t)} f_n^{c_1,c_2} & c_1 \neq c_2 \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

This matrix helps determine which clients' requests should be kept in the cache longer, based on the following event count for clients in the past.

Description of the Policy: When a new request (a_m, d_m, c_m) arrives, we first check whether it results in a cache hit or a cache miss. If it is a cache hit, the requested object is moved to the most recent position in the access order to reflect its recency. If it is a cache miss, the object is added to the cache as the most recent entry, and evictions are performed as required.

To determine which object(s) to evict at time a_m, we first update the matrix $F(a_m)$, which captures the number of following events between pairs of clients. Next, using this matrix, we calculate the maximum number of times that other clients have followed each client's requests. Finally, objects associated with clients having the fewest following events are evicted first. If

multiple objects correspond to clients with the same following event count, least recently accessed object is evicted. Note if no following events have been seen then the caching policy corresponds to LRU.

Remark 4.4 (Advantages of a Request-Based Window Over a Time-Based Window)**.** *Using a fixed request-based window of the last w requests per client, rather than a time-based window, provides a fair and consistent method for comparing following events across different clients. The key advantages of this approach are as follows:*

- ***Fair comparison across different request rates:*** *A request-based window ensures that following events are counted in a comparable manner for all clients. In contrast, a time-based window may count more events for clients with higher request rates, overestimating their influence on caching decisions.*
- ***Adaptability to traffic variations:*** *Since a request-based window directly tracks request behavior, it remains unaffected by fluctuations in request timing, making the caching policy more robust to dynamic traffic patterns.*
- ***Consistent eviction decisions:*** *By maintaining statistics based on a fixed number of past requests, a request-based window leads to stable and predictable cache eviction rules. In contrast, a time-based window varies with traffic conditions, potentially resulting in unpredictable cache behavior.*

4.3.2.2 Least Following and Recently Used with Smoothing (LFRUS (w, γ))

In the case that requests patterns are changing frequently, we consider a variant of LFRU policy, LFRUS, that adapts and assigns different weights to following events based on their recency. This policy is a combination of exponential averaging and sliding window. The key difference is in the calculation of $F^{c_1,c_2}(t)$, which is calculated as follows:

$$F^{c_1,c_2}(t) = \begin{cases} \left\lfloor \sum_{n=n^{c_2}(t)-w}^{n^{c_2}(t)} \gamma^{(n^{c_2}(t)-n)} f_n^{c_1,c_2} \right\rfloor & c_1 \neq c_2 \\ 0 & \text{otherwise.} \end{cases} \tag{4.25}$$

Here, γ is a parameter that determines the weight assigned to each following event, with more recent events given higher weightage. The notation $\lfloor x \rfloor$ represents the floor of x. This policy has two parameters - w, γ.

4.4 Simulation results

In this section, we evaluate the proposed caching policy and compare its performance against standard caching policies across different request traces. These traces range from grouped client request model to client requests for objects within a VR environment.

For the VR-based request traces, clients move in groups and request objects that fall within their visibility range. We consider two approaches to generating these requests. First, we simulate client movement within a toroidal space where data objects are randomly distributed, and all objects have equal size. Second, we use an actual VR environment where the size of data objects

depends on factors such as the number of vertices, edges, textures, and the distance of the client from the object.

The primary evaluation metric used in our analysis is the *cache hit ratio*, which measures the fraction of client requests that are successfully served from the cache.

Caching policies. We consider three additional online caching policies for evaluation.

- **Least Frequently Used (LFU):** LFU prioritizes caching data objects that have been accessed most frequently.
- **Belady:** Belady's algorithm evicts the data object that will be requested furthest in the future, making it an optimal but non-causal policy, as it requires full knowledge of the request sequence. We evaluate this policy for traces where all data objects have the same size.
- **Sieve** [71]: Sieve is a caching policy that retains recently accessed objects while efficiently managing evictions using a single queue and a "hand" pointer. Each object is marked as either visited or non-visited, and the least recently visited object with its visited bit unset is evicted. Similar to Belady, we evaluate this policy only for traces where all data objects have the same size.

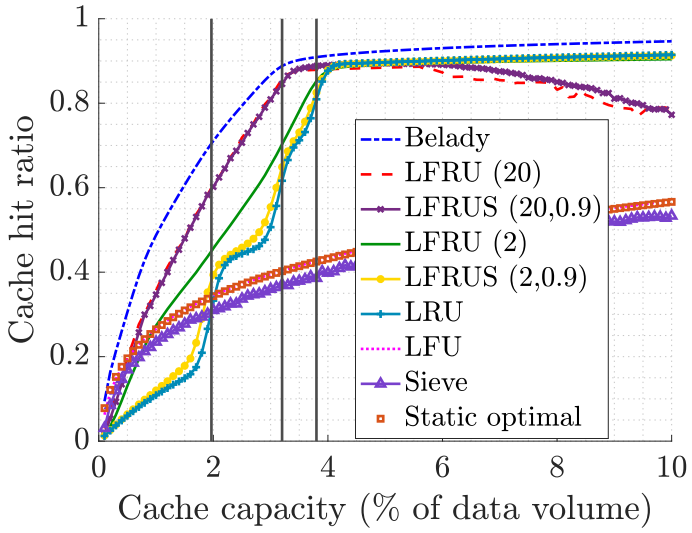


Figure 4.4: Cache hit ratio against cache capacity defined as percentage of trace foot print (the number of unique objects in the trace) under different caching policies.

4.4.1 Simulations for the grouped client request model

Setup. We consider a caching system with three groups ($G = 3$), where each group can request data objects from a distinct set. The sets of objects for the groups are defined as follows:

$$\mathcal{D}^1 = \{1, 2, \ldots, 1000\}, \quad \mathcal{D}^2 = \{1001, 1002, \ldots, 2000\},$$
$$\mathcal{D}^3 = \{2001, 2002, \ldots, 3000\}.$$

Each group consists of one leader and several followers. The number of followers in each group is:

$$f^1 = 8, \quad f^2 = 6, \quad f^3 = 4.$$

The delay between the i-th follower (for all $i \in \{1, \cdots, f^g\}$) and the leader of group g is modeled as constant and ordered, with the following values:

$$\text{Group 1: } \Delta_i^1 = 10i, \quad \text{Group 2: } \Delta_i^2 = 20i, \quad \text{Group 3: } \Delta_i^3 = 30i.$$

The probability of a request $p^g(d)$ for an object d in group g follows a Zipf distribution with parameters 0.8, 0.85, and 0.9 for Groups 1, 2, and 3, respectively.

Requests from group leaders arrive according to a Poisson process, with the following rates:

$$\lambda^1 = 10, \quad \lambda^2 = 15, \quad \lambda^3 = 20.$$

The size of all data objects is 1.

Results Discussion. Fig. 4.4 presents the cache hit ratio for different caching policies across various cache capacities, expressed as a percentage of the total data volume (computed as the number of data objects multiplied by their average size). The cache capacity ranges from small (0.1%) to large (10%). As expected, Belady's policy, which has full knowledge of future requests, performs the best.

For larger cache capacities, LRU and LFRU/LFRUS with $w = 2$ outperform other online policies such as LFU, LFRU/LFRUS with $w = 20$, and Sieve, as well as the offline Static Optimal policy. This is because, with a larger cache, objects remain in cache for extended periods. Consequently,

when a leader requests a data object, all its followers are more likely to experience a cache hit for the same object under LRU. However, the performance of LFRU/LFRUS with $w = 20$ degrades due to incorrect inferences of temporal causal relationships between clients. This issue arises for two primary reasons, first, leader requests are independent of past requests and follow a Zipf distribution, leading to frequent requests for popular objects, and second, as cache capacity increases, objects remain in the cache for longer, increasing the likelihood that a leader repeats a request for an object previously requested by another client. As a result, LFRU incorrectly infers that the leader follows that client. In contrast, with a smaller window size ($w = 2$), such incorrect inferences are quickly forgotten, minimizing their impact on caching decisions.

For small cache capacities, prioritizing data objects based on client-following behavior and request recency becomes crucial. In this case, LFRU/LFRUS effectively detects a limited number of following patterns and uses them to make more informed eviction decisions. This results in a significant performance gap between LFRU/LFRUS with $w = 20$ and $w = 2$ compared to LRU. A larger window size further widens this gap, as inferred following patterns persist longer, affecting eviction choices.

Under the grouped client request model, LRU performs well when the characteristic time, t_b^*—the time before a newly introduced object is evicted if there are no additional requests for this object—is longer than the delay between follower requests. In such cases, every follower request results in a cache hit, regardless of the requested object. To illustrate this effect, vertical

lines in Fig. 4.4 mark cache capacities of 2%, 3.2%, and 3.8% of the total data volume. These correspond to cache sizes where the characteristic time matches follower delays in Groups 1, 2, and 3, with delays of 10, 20, and 30 units, respectively. As expected, around these points, the LRU hit rate exhibits a sharp increase.

In contrast, under LFU, followers experience a cache hit only when they request the most frequently accessed data objects that remain in the cache. Additionally, due to the high number of distinct requests, LFU performs similarly to the Static Optimal policy. Similarly, Sieve prioritizes the retention of frequently and recently requested objects. However, as demonstrated in our results, these strategies are suboptimal in this setting because follower requests closely track leader requests, even for objects that are infrequently accessed overall.

While in this section, we examined client request patterns based on the Grouped Client Request Model, this model assumes that a leader's requests are independent of past requests. However, in practical scenarios, client requests are inherently influenced by movement and past interactions, particularly in environments such as VR. To better capture these real-world dependencies, in the next section, we shift our focus to a synthetic VR environment, where client requests arise as users navigate a shared space.

4.4.2 Simulations for synthetic request traces for client motion in a Toroid

4.4.2.1 Simulation environment for generating cache request traces

To analyze spatio-temporal patterns in data object requests, we generate cache request traces in a controlled simulation environment where clients move as part of distinct groups. The simulation takes place in a 3D toroidal cube, with each side measuring 1000 units. This space contains 4000 data objects, which are randomly distributed throughout the volume. The size of each data object is equal; we revisit this in the next section.

To model group motion dynamics, each group consists of a leader client and multiple follower clients. The total number of groups, the designated leaders, and the overall number of clients remain fixed throughout the simulation. Leaders follow unique motion paths, while followers trail behind with a fixed delay. Group composition can change over time, with followers either reordering themselves within the group or switching to a different leader.

The simulation operates in discrete time slots, where each time slot represents one unit of simulated time. During each time slot:

1. Leaders update their positions based on a constant speed of 25 units per time slot and their current direction. Every 10 time slots, each leader selects a new random direction. They move continuously through the toroidal space, reappearing on the opposite side when crossing a boundary.

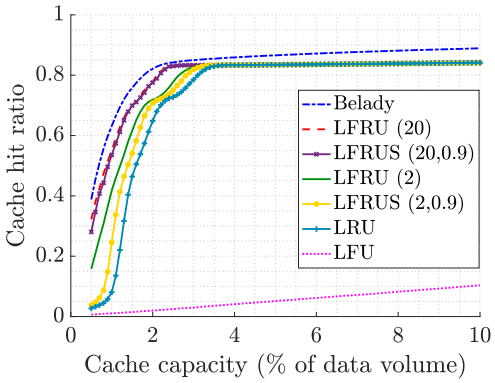


Figure 4.5: Cache hit ratio against cache capacity under different caching policies for Trace 1.

2. Followers update their positions by following the path of their assigned leader, but with a delay of a specified number of time slots. For example, if a follower has a delay of τ time slots, its position in the current time slot matches the leader's position from τ time slots earlier. This delay models how group members in the real world follow a leader with some lag.

3. Each client, including both leaders and followers, requests data objects located within a 360-degree field of view with a radius of 50 units.

The simulation runs for 10 million time slots, enabling the capture of long-term spatio-temporal patterns in client movement and data access.

4.4.2.2 Trace 1: Static following

Setup. In this trace, we simulate 3 groups comprising a total of 17 clients navigating the 3D toroidal space. Groups 1, 2, and 3 consist of 8, 4,

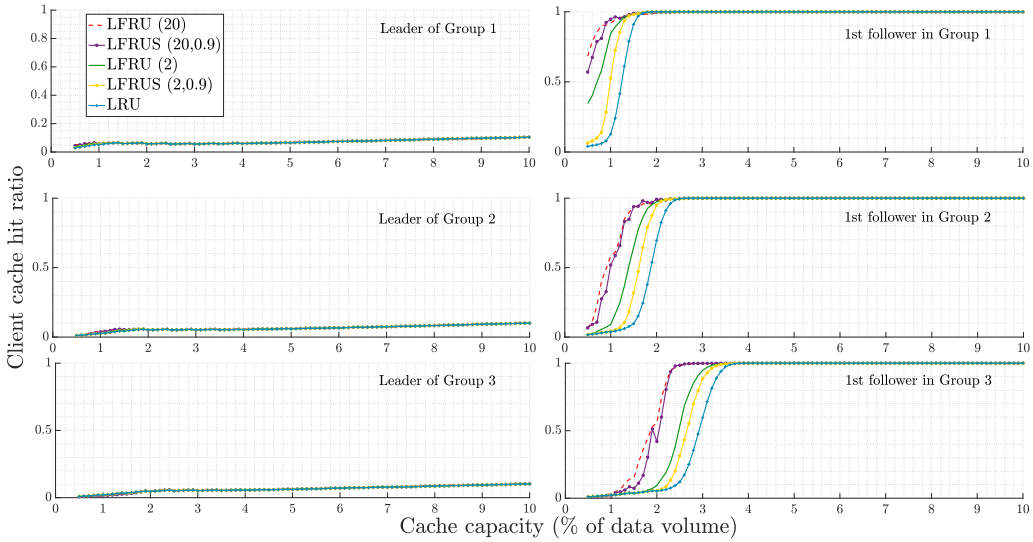


Figure 4.6: Cache hit ratio for different clients against cache capacity under different caching policies for Trace 1.

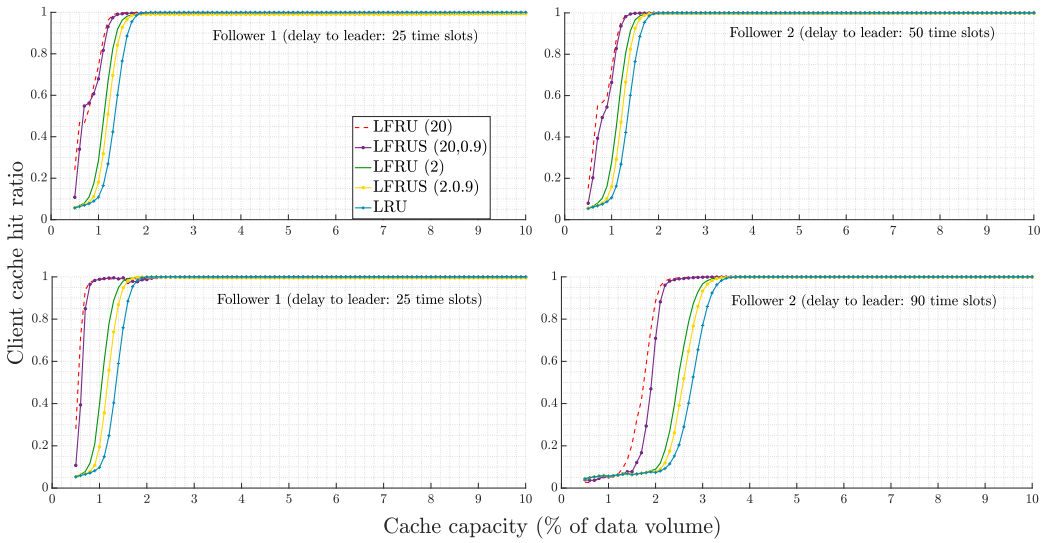


Figure 4.7: Comparison of client cache hit ratios across different caching policies under two delay configurations: uniform delays between followers (row 1) and non-uniform delays (row 2).

and 2 followers, respectively. Each group's leader is initialized at a random location and navigates according to the dynamics described above. The delay between the i-th follower and the leader varies by group. In Group 1, the i-th follower trails its leader with a delay of $4i$ time slots, e.g., Follower 1 trails the leader by 4 time slots, Follower 2 by 8 time slots, and so on. Similarly, in Group 2, the i-th follower trails its leader with a delay of $8i$ time slots, while in Group 3, the delay is $20i$ time slots. Each client maintains a local cache managed using the Least Recently Used (LRU) policy. The size of each local cache is set to 5% of b, where b is the total cache capacity of the edge or cloud server. When a client makes a request, it first checks its local cache. If the requested item is not found (a cache miss), the client forwards the request to the main cache (edge or cloud server). The cache hit ratio is defined as the fraction of requests to the main cache (edge or cloud server) that result in a hit.

Results discussion. Figure 4.5 shows the cache hit ratio for different caching policies at various cache capacities, expressed as a percentage of total data volume (computed as the number of data objects multiplied by their average size). The cache capacity ranges from small (0.1%) to large (10%). In Figure 4.6, we also show the client cache hit ratio for a selected set of caching policies. The client cache hit ratio is defined as the proportion of requests from a specific client that result in a cache hit, compared to the total number of requests. As expected, Belady's policy, which is based on the knowledge of future requests, performs the best. This is because it can retain objects that

will be requested again soon.

Next, LFRU/LFRUS with $w = 20$, LFRU/LFRUS with $w = 2$, and LRU show similar performance when the cache capacity is large. This happens because, with a larger cache, objects stay in the cache for longer periods. As a result, once a leader requests a data object, all its followers also experience a cache hit for that object, as shown in Figure 4.6. Therefore, even though our policies account for following events to decide on evictions, there is enough cache space to keep the objects requested by each leader.

However, when the cache capacity is small and space is limited, it becomes important to decide which objects to keep based on client-following behaviors and the recency of requests. This creates a noticeable performance gap between LFRU/LFRUS with $w = 20$ and $w = 2$ as compared to LRU. As seen in Figure 4.6 for our LFRU-based policies, followers start to experience cache hits for their requests, which does not happen with LRU. With a larger window, this gap becomes even larger, as any detected following event stays in the window for a longer period of time. However, if older following events are not given equal weight, a performance gap appears, as seen in the difference between LFRU and LFRUS with the same window sizes.

We also observe performance jumps for LRU. These jumps occur when the delay between followers in a group becomes roughly equal to the time a typical object stays in the cache under LRU. We can see this in Figure 4.6, where the client cache hit ratio for followers in different groups shows jumps in performance under LRU at different cache capacities. These jumps follow

the order of delays between followers in the groups, meaning that Group 1 followers, with lower delays, experience a higher cache hit ratio than Group 2 followers. Finally, the LFU policy, which prioritizes the most frequently accessed objects, performs poorly. This is because when a follower requests the same object as their leader, the object's access frequency increases, causing it to stay in the cache longer than needed. Instead, the policy should focus on keeping new data objects requested by group leaders, even if they are rarely accessed.

4.4.2.3 Effect of different delays between followers for Static following

Setup. In this simulation, we consider a group consisting of three clients: one leader and two followers. We examine two scenarios for the delay between followers - Case 1: The delay between followers is uniform. The i-th follower trails its leader with a delay of $25i$ time slots, Case 2: The delay between followers is non-uniform. Follower 1 trails its leader with a delay of 25 time slots, while Follower 2 trails its leader with a delay of 90 time slots.

Results discussion. Fig. 4.7 presents the client cache hit ratio for different caching policies at various cache capacities. In the first row, which corresponds to Case 1, both Follower 1 and Follower 2 exhibit similar performance. However, in the second row, which represents Case 2 with increased delay for Follower 2, we observe a decline in Follower 2's cache hit ratio compared to Case 1. This performance drop is consistent across all caching policies.

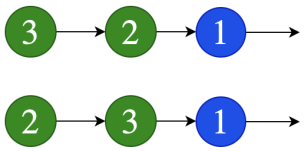
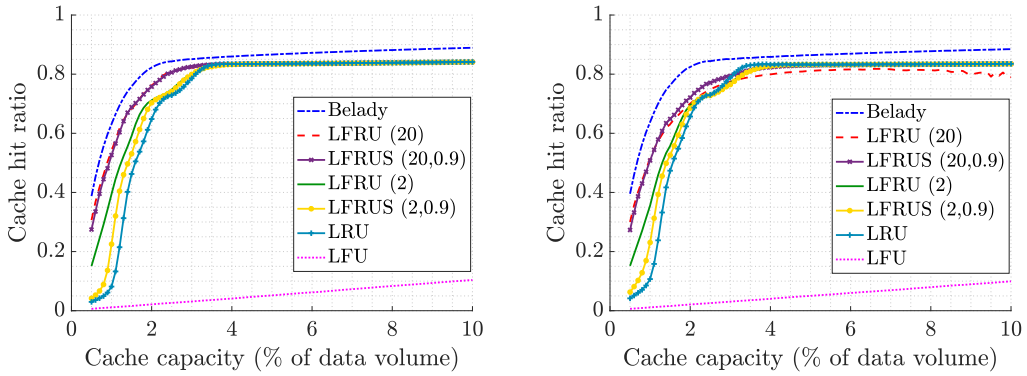


Figure 4.8: After every p time slots, Client 2 and 3 (followers) swap their positions in following Client 1 (leader).



(a) p = 1000. (b) p = 100.

Figure 4.9: Cache hit ratio against cache capacity under different caching policies for Trace 2.

4.4.2.4 Trace 2: Periodic order shuffling

Setup. In this trace, the group characteristics remain the same as in Trace 1. However, we introduce periodic shuffling of followers, denoted by a parameter p, which specifies the number of time slots in each period. At the start of a new period, follower $2i-1$ and follower $2i$ in each group exchange their delays/positions relative to the leader, see Fig. 4.8. This value of the parameter p allows us to evaluate the impact of varying the frequency of these swaps on the performance of our caching policies. As before, each client has a

local cache managed using the Least Recently Used (LRU) policy. The size of each local cache is set to 5% of b, where b represents the total cache capacity (of edge/cloud server).

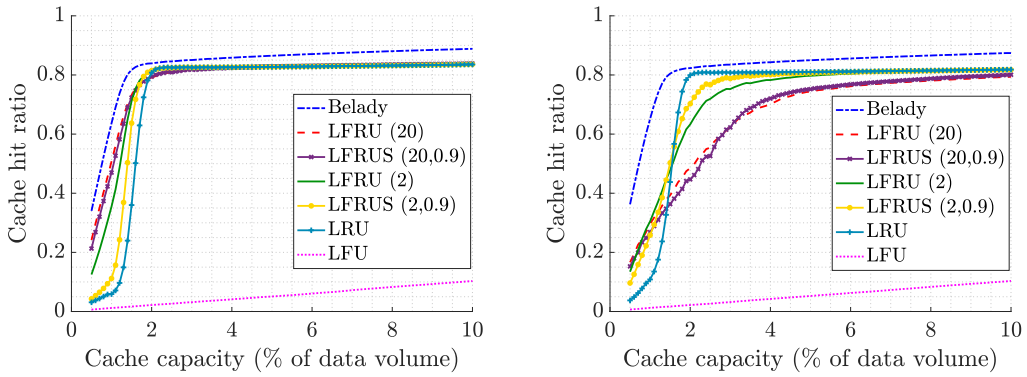
Results discussion. Figures 4.9a and 4.9b show the cache hit ratio for various caching policies when the period p is large and small, respectively, indicating how long a particular following behavior persists.

When p is large, the proposed caching policies that utilize inferred causal relationships continue to perform well. However, there is a slight performance degradation (though not noticeable) compared to Trace 1.

When p is small, the performance of LFRU with $w = 20$ decreases for larger cache sizes as compared to Trace 1 in the previous section. This happens because followers change their order more frequently, so the policy is biasing its decisions on incorrect inferences of following relationships. Additionally, the cache needs to flush out all data objects associated with clients that were previously perceived as being followed by others, but are no longer followed. This leads to a performance loss.

This issue can be mitigated by using a smaller window, as shown by LFRU with $w = 2$. A smaller window helps the policy adapt more quickly by forgetting past following events. Alternatively, assigning weights to following events ensures that only persistent following events influence the decision of which data objects to retain; see LFRUS variants of LFRU.

For small cache capacities, larger windows remain preferable because



(a) p = 1000. (b) p = 100.

Figure 4.10: Cache hit ratio against cache capacity under different caching policies for Trace 3.

the likelihood of LFRU detecting all following patterns is lower. As a result, even when the order changes, there is no significant performance degradation.

4.4.2.5 Trace 3: Periodic leader switching

Setup. In this trace, the number of leaders remains 3, and the total number of clients remains 17. However, we introduce periodic leader changes for followers, controlled by a parameter p, which specifies the number of time slots in each period. At the start of a new period, each follower selects a new leader with predefined probabilities: Leader 1 is chosen with probability 0.5, Leader 2 with probability 0.3, and Leader 3 with probability 0.2. When a follower selects a new leader, its delay relative to this new leader is determined by the number of followers who have already selected this leader and the delay between consecutive followers, which we set to 5. Consequently, the i-th

follower trails behind its newly selected leader by $5i$ time slots. As before, the parameter p allows us to study the effect of varying the frequency of these leader switches on the performance of our caching policies. Lastly, each client has a local cache managed using the Least Recently Used (LRU) policy. The size of each local cache is set to 5% of b, where b represents the total cache capacity (of edge/cloud server).

Results discussion. Figures 4.10a and 4.10b show the cache hit ratio for various caching policies when the period p is large and small, respectively. The period p indicates how long a group configuration and the order of followers in that group persist.

When duration is large, as shown in Fig. 4.9a, our caching policies that use inferred causal relationships continue to perform well.

When the duration is small, there is a significant performance gap between LFRU with $w = 20$ and LRU for medium to large cache sizes. This occurs because, in this case, followers not only change their order, but also switch leaders. With a larger window, it takes more time to forget and update the cache. This performance gap decreases when using a smaller window, such as LFRU with $w = 2$.

However, the best performance is achieved when we also apply smoothing along with a small window. This allows the policy to adapt quickly and giving more weightage to recent following events between clients. This approach also shows a significant performance improvement over LRU, especially

Figure 4.11: The virtual city of the simulation.

for smaller cache capacities.

The synthetic VR environment in this section captures how client requests depend on movement, but does not account for certain real-world factors. In practice, objects in a VR scene may be occluded by other objects, making them temporarily inaccessible. Additionally, the size of an object in a request depends on its pixel size, texture, geometric complexity, and distance from the client. These factors affect how objects are rendered and influence caching decisions. In the next section, we refine our request model to incorporate these additional constraints, making it more representative of real-world VR environments.

4.4.3 Simulations for emulated VR request traces

4.4.3.1 Simulation environment for generating cache request traces

We generated cache request traces using a simulator that models a virtual desert city, measuring 376.29 $\times$ 608.22 m^2 and containing 1,176 objects (e.g., buildings, trees, fountains). In this simulation, a virtual reality (VR)

client navigates the city from a height of 1 to 2 meters above the ground, which represents the typical height of a VR headset, whether the client is sitting or standing. Objects in the city may be occluded by buildings depending on the client's position, so clients can only request visible data.

To determine visibility, we dynamically compute the set of objects that are visible from all directions around the client's current position. This approach is preferred over relying on a fixed or random view direction, as it ensures that we send data for all possible directions rather than waiting for the client to specify their view. This method is particularly important because visibility computation is computationally expensive, and the client's viewpoint may change rapidly. Computing visibility only for the current view direction would be inefficient, especially when the client may quickly turn or move.

Clients navigate through alleys along pre-designated, looped paths, avoiding collisions with objects, as shown in Fig. 4.11. In our dataset, there are always three client groups, each consisting of five clients and each group follows a distinct path. Within each group, one client serves as the leader, while the remaining clients follow in the same direction. These paths are bidirectional and non-intersecting, and all clients move at a constant speed of 2 m/s.

The simulation operates in discrete time slots. In each slot, each client updates its position based on the elapsed time since the previous slot and recomputes the visibility of objects. A client initiates a request for an object if it becomes visible in the current time slot and was not requested in the

previous slot. This request mechanism assumes that the client's local cache is large enough to store data retrieved in a given time slot for use in the subsequent slot.

In the simulation, data objects come in multiple versions, each varying in size. The version of a requested data object depends on the distance of the client from the object: If the client is within 10 m, it requests the highest quality version; if it is beyond 50 m, it requests lowest quality version; and for distances between 10 and 50 m, it requests middle quality version. The size of each version is influenced by factors such as pixel resolution, texture, and geometric detail. For example, the highest quality version of an object may be 1MB, the middle quality version 0.5MB, and the lowest quality version 0.1MB. Additionally, each request must be served with the exact version specified by the client; a request for one version cannot be fulfilled using another version of the same object.

4.4.3.2 Trace 1: Unstructured follower requests.

Setup. In this trace, all clients within the same group are initialized near a common point on their designated path (Fig. 4.12). Each client starts at a random position within 4 meters of this point. Clients do not enter buildings during navigation.

All followers move in the same direction as their group's leader, maintaining a constant delay of 0.5 seconds. Specifically, if the leader changes direction at a corner, all followers adjust their movement 0.5 seconds later.

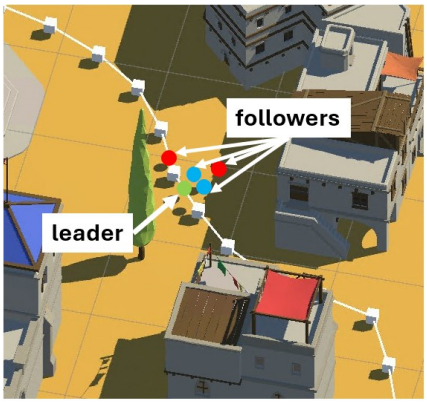


Figure 4.12: Example relative positions of the leader and followers in unstructured follower requests.

Each client maintains a local cache of previously requested objects managed using the LRU policy. The size of each local cache is set to 5% of b, where b is the total cache capacity of the edge or cloud server. When a client makes a request, it first checks its local cache. If the requested item is not found (a cache miss), the client forwards the request to the main cache (edge or cloud server). We have considered a local cache for each client as described for all simulation results discussed hereafter.

Results discussion. Fig. 4.13 presents the cache hit ratio for different caching policies across various edge/cloud cache capacities, expressed as a percentage of the total data volume (computed as the number of data objects multiplied by their average size). The cache capacity varies from small (1%) to large (22%).

Our results show that LRU performs best for moderate to large cache capacities. This is because, in unstructured following, followers are distributed

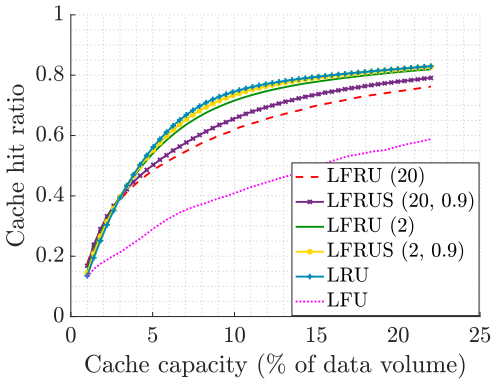


Figure 4.13: Cache hit ratio against cache capacity under different caching policies for unstructured follower requests.

around the leader in a staggered pattern and do not consistently request the same set of data objects. Consequently, the number of consistent following events is lower, making inferred temporal relationships less effective for cache management. In such cases, eviction decisions based on recency outperform those based on inferred causal relationships. This is evident from the performance gap between LFRUS and LFRU for the same window size, where smoothing enables the caching policy to assign greater importance to recent following events. Additionally, using a smaller window size improves adaptability by allowing the policy to quickly discard outdated following patterns. For small cache capacities, LFRU/LFRUS performs better with larger window sizes. Rather than considering the recency of all client requests, these policies prioritize clients that are closer and exhibit more persistent following behavior, leading to an improved cache hit ratio.

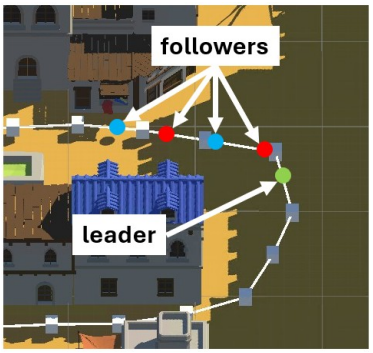


Figure 4.14: Example relative positions of the leader and followers in structured follower requests.

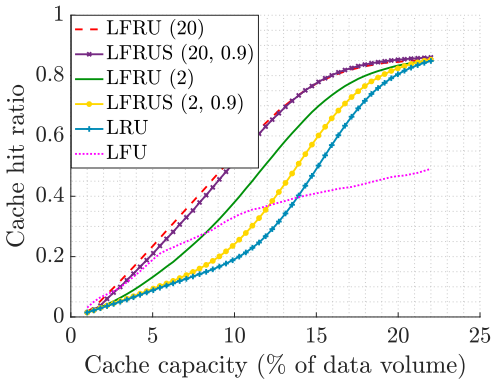


Figure 4.15: Cache hit ratio against cache capacity under different caching policies for structured follower requests.

4.4.3.3 Trace 2: Static following (Structured follower requests)

Setup. In this trace, all followers move along the path by following their leader (Fig. 4.14). The leader is initialized at a random point on the path, and each follower is subsequently initialized with a constant delay of 2 seconds.

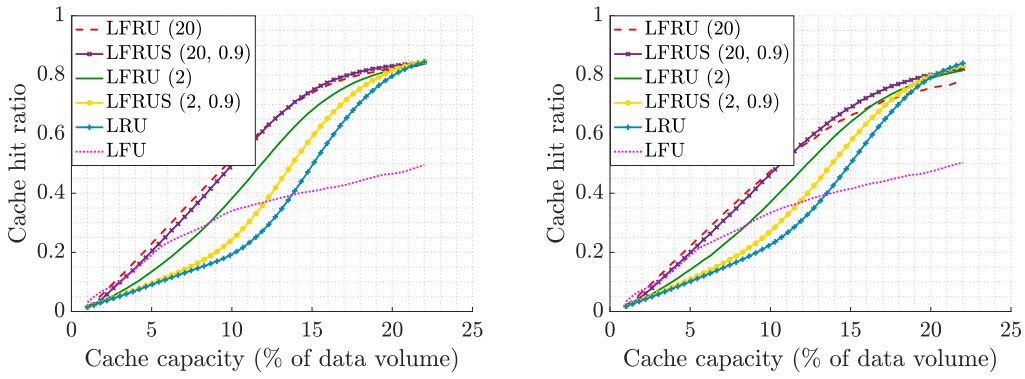
For example, if the leader is initialized at position x, the first follower

starts at x - 4 meters along the path (assuming movement in a positive direction). The second follower is initialized at x - 8 meters, and so forth. Throughout the simulation, all clients maintain their initial relative distances from one another on the path.

Results discussion. Fig. 4.15 presents the cache hit ratio for different caching policies across various cache capacities, ranging from small (1%) to large (22%).

Our results indicate that LFRU/LFRUS with $w = 20$ consistently outperforms all other caching policies across all cache capacities. This is because, in structured following, followers sequentially request data after their leader and tend to repeat the leader's requests. Once these following events are detected, they enhance cache eviction decisions by prioritizing the retention of objects requested by a client (excluding the last follower) within a group, as these objects are likely to be requested again by another follower in the same group.

The advantage of retaining the following events detected for longer is evident in the performance gap between LFRU with $w = 20$ and $w = 2$. A larger window size allows the policy to leverage persistent following patterns more effectively, leading to higher cache hit ratios. Furthermore, in this setting, smoothing is unnecessary, as following events remain stable over time.



(a) p = 150 seconds.

(b) p = 30 seconds.

Figure 4.16: Cache hit ratio against cache capacity under different caching policies for Periodic order shuffling.

4.4.3.4 Trace 3: Periodic order shuffling

Setup. In this trace, the setup remains the same as in Trace 2, except that followers periodically swap positions in pairs. Every p seconds (30s or 150s), adjacent followers swap positions: the first follower swaps with the second, the third with the fourth, and so on.

The swap occurs gradually over a duration of 3 seconds, rather than instantaneously. During this process, clients continue their normal movement along the path, but their relative distances temporarily change as they transition to their new positions.

Results discussion. Figures 4.16a and 4.16b present the cache hit ratio for different caching policies when the period p is large and small, respectively. The period p represents the duration for which a particular following behavior persists.

When p is large, caching policies that leverage inferred causal relationships maintain strong performance. However, a slight performance drop is observed compared to Fig. 4.15, though it remains negligible.

When p is small, the performance of LFRU with $w = 20$ deteriorates for larger cache sizes compared to scenarios with a larger p. This decline occurs because followers change their order more frequently, leading the policy to make incorrect inferences about following relationships. Additionally, the cache must evict objects associated with clients who were previously identified as being followed but are no longer, resulting in performance degradation.

This issue can be mitigated by using a smaller window size, as demonstrated by LFRU with $w = 2$. A smaller window enables the policy to adapt more quickly by discarding outdated following events. Alternatively, assigning weights to following events ensures that only persistent relationships influence caching decisions, as seen in the LFRUS variants of LFRU.

For small cache capacities, larger window sizes remain beneficial, as LFRU is less likely to capture all following patterns. Consequently, even when follower order changes, the performance does not degrade significantly.

4.5 Conclusion

In this chapter, we introduced a model for capturing correlated client request patterns. We then showed that correlations in client requests influence the characteristic time of an LRU-managed cache, leading to improved cache

hit ratios compared to LFU as cache size increases. To further leverage these correlations, we proposed LFRU, an adaptive caching policy that dynamically infers and exploits causal relationships between client requests. By incorporating both the recency of requests and the frequency with which clients follow others' requests, LFRU enhances its eviction decisions. Our empirical evaluations on synthetic and VR-based datasets demonstrated that LFRU achieves up to 2.9× and 1.9× improvements in cache hit ratio over LRU and LFU, respectively, under structured following. These findings underscore the importance of incorporating request correlations in caching strategies and provide a foundation for the development of more adaptive and efficient caching mechanisms in future networked systems.

Chapter 5

Optimal Scheduling Algorithms for LLM Inference: Practice

This chapter[1] looks at serving large language model powered chatbot requests with heterogeneous Service Level Objectives (SLOs) on a single GPU. We devise a practical scheduler, SLO Aware LLM Inference (SLAI) scheduler that supports such requests and empirically demonstrate its performance on NVIDIA RTX ADA 6000.

[1]This chapter is based on the work to be submitted to Sigmetrics 2026:

- A. Bari*, P. Hegde*, G. De Veciana, "Optimal Scheduling Algorithms for LLM Inference: Theory and Practice," *Submitted to Sigmetrics 26*

Agrim Bari led the formulation of the practical problem, the design of the policy, execution of experiments, and the writing of the paper.

5.1 Introduction

LLM inference systems. The core problem in Large Language Model (LLM) inference is to generate a response autoregressively, one token[2] at a time, given a *prompt*—for example, "What is the capital of France?" producing the output "It is Paris." Modern LLMs such as GPT-4, Llama 3, and Gemini now power a wide range of services, including chatbots, coding assistants, and search engines. These services handle millions of user requests daily, and private deployments are rapidly increasing. As a result, there is growing interest in optimizing how requests are processed across one or more Graphics Processing Unit (GPU)-enabled nodes in data centers, since improved efficiency can lead to significant reductions in infrastructure and operating costs.

Objectives for LLM serving systems. To meet growing demand, LLM systems must be carefully designed to make efficient use of hardware. A well-designed system keeps each active GPU busy—fully utilizing both its compute and memory—while also keeping response times low. This leads to two main goals: (1) achieving high throughput, measured in requests per second, to reduce the cost per request, and (2) maintaining low latency, which directly affects user experience. Latency is typically measured using two Service-Level Objectives (SLOs)[3]: *Time To First Token* (TTFT), which is the delay between a request's arrival and the generation of the first output token, capturing how

[2]A token in a LLM is a unit of text-such as a word, subword, or character-used as the basic input element for processing and generation.

[3]Thresholds may vary by application, but these two metrics are commonly used.

long a user waits before the LLM starts responding; and *Time Between Tokens* (TBT), which is the time between successive output tokens, indicating the rate at which the response is streamed to the user. In real-world systems, request routing, scheduling, and caching are used to meet these goals. This paper focuses on *scheduling*, which plays a critical role in increasing throughput, reducing TTFT, and keeping TBT within acceptable limits.

Phases of an LLM request and scheduler decisions. Each request to an LLM based on the Transformer architecture goes through two main phases: *prefill* and *decode*. In the *prefill-phase*, the model processes the entire prompt and generates the first output token. After that, the request enters the *decode-phase*, where it produces one token at a time in an auto-regressive manner until a `stop` token is generated. These two phases have distinct characteristics. The prefill-phase is highly parallelizable and can fully utilize the GPU's compute resources. By contrast, the decode phase is sequential and has low parallelism, which means that multiple decode-phase requests must be batched together to make efficient use of the GPU. Additionally, Transformer models store intermediate representations of tokens—called the *Key-Value (KV) cache*—which grow with the number of tokens processed and consume GPU memory. The scheduler's job is to select a mix of prefill-phase and decode-phase requests to include in each GPU batch. These decisions must balance GPU compute and memory bandwidth usage, stay within the memory budget, and meet latency SLOs.

Challenges in scheduler design. Designing an effective scheduler for LLM inference presents several key challenges. First, GPU compute efficiency depends heavily on the composition of each batch—that is, the mix of prefill and decode-phase requests scheduled together. As a result, the scheduler must construct batches carefully to make the most of available resources. Second, the scheduler must balance resource allocation between the prefill and decode phases, as both phases have their respective SLOs: TTFT and TBT. Prioritizing prefill-phase requests can reduce TTFT but may delay decodes and worsen TBT. Conversely, prioritizing decode-phase requests keeps TBT low but can lower compute utilization and increase TTFT for new requests. Third, while the prompt length is known upon a request arrival the output length is unknown, making memory management complex—particularly since GPU memory is often a bottleneck. Finally, many LLM serving systems support multiple user tiers which have heterogeneous performance needs.

Our approach. In this work, we approach the LLM scheduling problem from two complementary perspectives. From a theoretical standpoint, we develop a rigorous framework for analyzing and achieving throughput-optimal scheduling. From a practical perspective, we design a scheduler that dynamically adapts to diverse latency SLOs across heterogeneous user tiers.

Contributions. Our key contributions are:

1. *A SLO-Aware Scheduler.* We design a practical scheduler called SLO-Aware LLM Inference (SLAI) scheduler that aims to minimize online me-

dian TTFT when serving requests with heterogeneous TBT constraints. To achieve this, SLAI uses online measurements to decide when the execution of a decode-iteration has become *time-critical*, i.e., should be prioritized for scheduling. In addition, it uses the known prompt length information to order prefill-phase requests so as to reduce the median TTFT. See Section 5.2.

2. *Experimental Performance.* We evaluate SLAI on the `openchat_shareGPT4` dataset using the Mistral-7B LLM on an NVIDIA RTX ADA 6000 GPU. Our results show that SLAI can reduce the median TTFT by 53% while meeting TBT requirements compared to Sarathi-serve, the current state-of-the-art scheduler. Additionally, when median TTFT can not exceed 0.5 seconds, SLAI increases the serving capacity by 26%. See Section 5.3.

5.1.1 Related Work

LLM serving systems must make decisions about *when* to run the prefill and decode phases of a request, *where* to execute each request, and *how* to manage the growing KV cache produced by the model. Based on these challenges, prior work can be broadly categorized into three areas: (a) scheduling within a single inference node, (b) routing across multiple inference nodes, and (c) managing the KV cache.

5.1.1.1 Scheduling policies

Schedulers differ in how they prioritize prefill-phase and decode-phase requests, and in their batching granularity. Below, we highlight some of the key work in this literature.

Decode-prioritizing (request-level) schedulers. Frameworks such as FasterTransformer [51, 52] and the request-level mode of TensorRT-LLM process a set of requests till completion, before admitting any new requests. Since new prefill-phase requests never interrupt ongoing decode-phase requests, these schedulers perform well on TBT. However, they have low throughput when there is a imbalance in the total (prompt and output) length of requests and thus GPU may be under-utilized.

Prefill-prioritizing (iteration-level) schedulers. Iteration-level batching, first introduced by Orca [67], enables dynamic admission and completion of requests at each forward pass. However, it relies on static memory allocation for the KV cache, which limits the number of concurrent requests to 16 on an A100 GPU. vLLM [41] overcomes this limitation using paged attention, allowing more flexible memory management and increasing the maximum number of concurrent requests to 128. Additionally, like FlashDecoding++ [24] and DeepSpeed-FastGen [23], vLLM aggressively admits new prefill-phase requests to improve throughput. However, this eager admission policy can delay decode iterations—especially for long prompts—leading to higher TBT latencies.

Hybrid schedulers. Sarathi-Serve [1] introduces a token-budgeted, chunked-prefill strategy to balance throughput and TBT, effectively reducing decode-iteration stalls that are common in prefill-prioritizing schedulers. Beyond such scheduling-focused methods, recent systems propose orthogonal techniques aimed at improving overall LLM inference performance. BlendServe [72] targets offline workloads by reordering requests based on their resource usage profiles to improve hardware efficiency. POD-Attention [35] enables pipelined execution of prefill and decode phases to increase kernel overlap and improve GPU utilization. HydraGen [34] reduces redundant computation by identifying and merging shared prompt prefixes across requests. DistServe [74] adopts a disaggregated architecture that separates prefill and decode execution across different nodes, thereby eliminating intra-GPU contention; however, it introduces communication overhead due to the transfer of large KV caches.

5.1.2 Cluster-level Routing

Per-GPU schedulers rely on the router to provide a well-balanced stream of requests. Most production systems still use simple strategies like round-robin or shortest-queue routing. These methods overlook the complex relationship between request length, prompt size, and current GPU state. The Intelligent Router for LLM Workloads [27] addresses this by framing routing as a sequential decision problem. It trains a workload-aware reinforcement learning agent to minimize overall latency by predicting each request's re-

sponse length and estimating how much delay each placement would cause.

5.1.3 KV-Cache Management

In transformer models, self-attention reuses all previous tokens, making KV-cache management critical for both speed and capacity.

Memory layout. vLLM [41] uses paged attention, which divides GPU memory into fixed-size blocks that are dynamically assigned to requests. This reduces fragmentation and allows hundreds of requests to run in parallel. Llumnix [59] builds on this by migrating KV tensors across replicas in real time, balancing memory usage and lowering preemption costs.

Prefix reuse and compression. Some systems try to reduce the amount of KV data stored or recomputed. SGLang [73] introduces a radix-tree cache and orders batches to maximize prefix reuse across multi-turn chats and speculative decoding. CacheGen [45] compresses KV blocks and streams them on demand, while FlashInfer [65] creates custom GPU kernels that operate directly on the compressed format. These techniques are independent of scheduling and routing and can be used alongside paged layouts or distributed setups.

Preemption strategies. When GPU memory runs out, systems must either recompute or offload paused requests. vLLM [41] and Sarathi-Serve [1] evict stalled requests, splice their outputs back into the prompt, and later rebuild the KV cache. DistServe [74], on the other hand, moves the KV tensors to host memory and resumes decoding once space is available. Each method

involves a trade-off between memory traffic and computation, and interacts closely with the scheduler's design.

5.1.4 Speculative decoding

Speculative decoding [43] offers a complementary approach that accelerates decode-phase computation by generating token drafts using a smaller auxiliary model, which are then validated by the larger target model. While originally proposed to reduce per-request latency, this technique can also benefit scheduling by reducing decode durations, improving GPU throughput, and enabling more efficient batch formation under tight latency constraints.

5.2 SLO Aware LLM Inference Scheduler

Request classes. In real-world LLM serving systems, requests may come from different classes of user with heterogeneous latency SLOs. For example, paying users typically expect fast and smooth responses, especially during token generation, which requires stricter TBT deadlines. By contrast, free-tier users are generally more tolerant of delays and can be served with more relaxed TBT constraints. Managing these mixed latency requirements well is important to keep users satisfied while making efficient use of system resources.

Recall that we consider a class of schedulers that executes prefill-phase and decode-phase requests as sequences of prefill-iterations and decode-iterations, respectively.

Motivation. The throughput-optimal RAD scheduler described in the theoretical section of the paper focuses on maximizing throughput, but it does not consider latency SLOs during either the prefill or decode phases due to the challenges mentioned in practical insights section in the paper. However, in practice, meeting these latency constraints is critical to deliver a good user experience.

Sarathi-Serve, the current state-of-the-art scheduler, addresses this by chunking long prefill-phase requests into smaller chunks and interleaving them with decode-phase requests in each batch. Each batch is constrained by a token budget—the maximum number of tokens it can process. Sarathi-Serve includes all active decode-phase requests from the previous batch in the current one and uses the remaining token budget to schedule prefill-iterations. However, it treats all decode-iterations of associated decode-phase requests as if they had the strictest TBT deadline, even when actual TBT deadlines vary across requests. While this conservative strategy ensures tail TBT latency is below some threshold, it can lead to inefficient use of batch capacity and does not address reducing TTFT for prefill-phase requests. To overcome this limitation, we propose the SLO-Aware LLM Inference (SLAI) Scheduler. SLAI tracks each decode-iteration's TBT deadline and delays its inclusion in a batch until necessary. This allows the scheduler to allocate more of the batch's token budget to prefill-iterations earlier, without missing TBT deadlines on decode-iterations. As we will show, this approach better aligns scheduling decisions with request-specific needs, resulting in lower median TTFT for prefill-phase

requests while still meeting tail TBT latency constraints on decode-iterations.

Key concepts and parameters. We begin by explaining how SLAI dynamically decides when to include a decode-iteration in a batch. The key idea in this process is the *last schedulable time*, which determines when a decode-iteration becomes critical and must be included to meet its latency target.

Let $\delta > 0$ be an offset parameter for SLAI that defines how early a decode-iteration should be considered critical. Consider the i^{th} decode iteration for a given request j which has an SLO requirement of TBT_j, SLAI notes the end time of the most recent batch in which its $(i-1)^{\text{th}}$ decode-iteration was included, denoted $e_{i-1,j}$. It also maintains the running average of batch execution times observed so far, denoted by $\bar{t}_{\text{batch}}$. We define its *last schedulable time* as:

$$C_{i,j} = e_{i-1,j} + \text{TBT}_j - \delta \cdot \bar{t}_{\text{batch}} \tag{5.1}$$

This is the latest wall-clock time by which decode-iteration j must be included in a batch to meet its TBT deadline. When constructing batch m at time t_m, the scheduler checks each active (in progress that currently occupies GPU memory) decode-phase request and labels its decode-iteration as *critical* if $t_m \geq C_{i,j}$; otherwise, it is considered as *non-critical* and can be deferred to a later batch.

Besides this dynamic prioritization, SLAI uses several key parameters to balance latency targets with efficient GPU use. The token budget τ sets the

maximum token count allowed in a batch, ensuring effective use of the GPU's compute resources without violating TBT SLOs. A cap on the number of active requests α limits how many active requests are allowed at once, helping prevent memory overflows for large models or long prompts. A decode limit β restricts how many of decode-iterations can be included in a batch, avoiding long batch execution times due to too many decode-iterations in a batch. Finally, the offset parameter δ provides a safety margin for the last schedulable time computation to absorb variability in batch execution and reduce the risk of missing TBT SLOs. We will later discuss how these parameters interact and influence the behavior and performance of the scheduler. Next, we describe how the SLAI scheduler works.

Batch construction. We now describe how a batch is constructed by the SLAI scheduler. At each decision point, the scheduler forms a batch from the set of active requests and new requests that have not yet been processed. While forming a batch, it must respect several system constraints: the token budget (τ), the cap on active requests (α), and the decode request limit (β). The batch construction follows these steps:

1. *Identify critical decode-iterations: For each active decode-phase request, compute the last schedulable time for its decode-iteration. A decode-iteration is marked as* critical *if current time is past its last schedulable time; otherwise, it is marked as* non-critical.

2. Add critical decode-iterations: Include critical decode-iterations in the

batch, in the increasing order of last schedulable time. This ensures that decode-iterations that are closest to their TBT deadline are scheduled first.

3. *Add prefill-phase requests: Next, add prefill-phase requests in a non-preemptive manner. Among these, requests that have already been scheduled at least once (i.e., active prefill-phase requests) are given a higher priority. If token budget and cap on number of active requests has not been exceeded, new prefill-phase requests are considered. We consider two possibilities for ordering the incoming requests: Shortest Prefill First (SPF) to reduce the average or median TTFT or First Come First Serve (FCFS) order to ensure fairness.*

4. Add non-critical decode-iterations: Finally, if there further token budget remains and number of decodes in the batch are less than the decode limit, include additional non-critical decode-iterations in increasing order of their last schedulable time.

Next, we discuss the impact of parameters other than the offset (δ), which has already been covered in our earlier discussion of the scheduler.

5.2.1 Impact of different scheduler parameters

5.2.1.1 Token budget (τ).

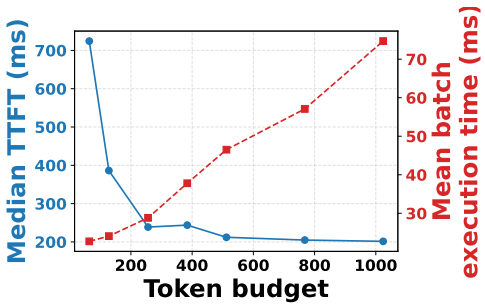
The token budget places an upper limit on the number of tokens that can be processed in a single batch—one token per decode-phase request and

$c \geq 1$ tokens per prefill chunk. Fig. 5.1a shows how the choice of token budget τ affects TTFT and batch-execution time. When τ is *small*, a 2048-token prefill must be split into many small chunks. This leads to frequent kernel launches and synchronization, causing the GPU to spend more time idling. As a result, TTFT increases. Conversely, when τ is *large*, the scheduler can process the entire request in fewer large batches. This improves GPU utilization but each batch takes longer to execute. If a decode-iteration is scheduled during such a long batch, it must wait, increasing the risk of violating its TBT constraint. The scheduler must thus choose a token budget τ that balances efficiency and responsiveness. Batches should be large enough to use the GPU effectively, but not so long that they excessively delay latency-sensitive decode-phase requests. Choosing this value carefully is a key part of designing an effective scheduling policy.

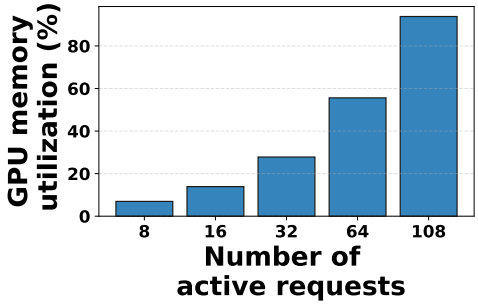
5.2.1.2 Cap on the number of active requests (α).

Each request generates KV tensors that must be stored in GPU memory until the request is completed. Fig. 5.1b shows how GPU memory utilization grows with the number of active decode-phase requests, based on runs with N concurrent decode-phase requests (each with a 2048-token prefill followed by a 1-token decode iteration) on Mistral-7B. As more requests become active, memory usage increases steadily.

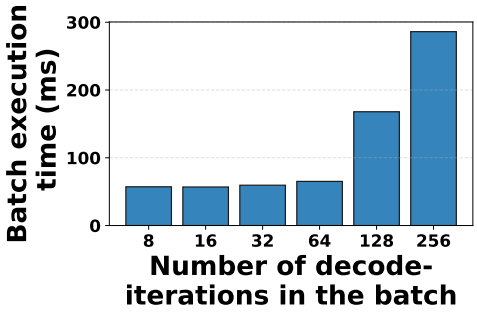
When too many requests are active, the scheduler may need to evict KV tensors to make room for new ones. If a request's KV tensors are evicted,



(a) Median TTFT and mean batch execution time for 100 independent requests (each with a 2048-token prefill and 1-token decode) as a function of the token budget.



(b) GPU memory usage versus number of concurrent decode-phase requests (each with a 2048-token prefill and 1-token decode).



(c) Batch execution time for N decode-phase requests (token position 513), co-scheduled with one prefill-only request to fully utilize the token budget of 512 tokens.

Figure 5.1: Impact of token budget, concurrency, and batch composition on request execution latency and GPU memory usage for Mistral-7B on a single NVIDIA RTX ADA 6000 GPU.

they must be recomputed before the request can resume decoding. This adds unnecessary delay and increases TTFT, even for requests that have not yet been scheduled. To avoid this, the scheduler should cap the number of active requests, ensuring that all necessary KV tensors can remain in memory without eviction. Doing so helps maintain low latency and avoids unnecessary recomputation overheads.

5.2.1.3 Decode limit (β).

The decode limit sets an upper bound on how many decode-iterations can be included in a single batch. Fig. 5.1c shows how batch-execution time changes as the number of decode-iterations increases, while keeping the token budget fixed at 512. When only a few decode iterations are present, the batch finishes quickly. However, as more decode-iterations are added, the batch takes significantly longer to finish due to increased pressure on compute and memory resources. Each decode-iteration triggers self-attention computation, which involves GeMVs per request—an inefficient computation on GPU.

When many decode-phase requests are active, limiting the number of decode-iterations in each batch helps control latency. In order to meet strict TBT deadlines, the scheduler can cap the number of decode-iterations per batch, reducing the number of TBT violations and maintaining better responsiveness under load.

5.3 Simulation results

Implementation. We built SLAI on top of the open-source implementations of Sarathi-serve [1] and vLLM [41].

Evaluation. We evaluate SLAI using the Mistral-7B [32] model and run all experiments on a single NVIDIA RTX ADA 6000 GPU. For our workload, we use the `openchat_shareGPT4` [61] dataset, which contains multi-round conversations between users and ChatGPT4 [53]. Each round is treated as a separate request.

Our baseline is Sarathi-serve configured with FCFS ordering for prefill-phase requests, referred to as Sarathi-serve (FCFS), which represents the current state-of-the-art in LLM inference scheduling on a single node. We also evaluate a variant of Sarathi-serve that uses shortest prefill first ordering, referred to as Sarathi-serve (SPF).

Similarly, we assess SLAI under both FCFS and SPF prefill orderings, referred as SLAI (FCFS, fixed offset) and SLAI (SPF, fixed offset), respectively. In addition, we evaluate a dynamic version of SLAI, called SLAI (SPF, dynamic offset), where the offset δ is adjusted at runtime based on GPU memory utilization measurements. When GPU memory usage is low, a small offset is used to allow more prefill-phase requests into the system. When memory usage is high, a larger offset is applied so that decode-iterations are marked critical earlier and prioritized accordingly, thus clearing out memory. For completeness, we also include vLLM in our comparisons, a scheduler that

prioritizes prefill-phase requests and serves as another relevant baseline.

Metrics. We evaluate two key metrics. The first measures the median TTFT since this is measured only once per request. This reflects how well the scheduler meets responsiveness objectives across requests. The second metric is the 99th percentile of the TBT, which is computed once per generated token and captures tail latency during the decode-iterations. This helps assess how smoothly tokens are generated over time.

Workload. To emulate realistic traffic, we generate synthetic traces based on request length distributions observed in the dataset. Prefill and decode lengths follow the distributions shown in Table 5.1, and request arrivals are modeled using a PPP. We cap each request's length to 8192 tokens in total. We consider two types of requests associated with paying and free-tier users. Paying users expect faster and smoother generation compared to free-tier users. To reflect this, we assign a TBT SLO threshold of 0.1 seconds for paying users and 0.5 seconds for free-tier user. These values are slightly relaxed compared to real-world production settings because our implementation is in Python (which is not fully optimized), includes telemetry overhead, and also reflects the inherent performance limitations of the model–hardware combination. Each incoming request is randomly marked as associated with a paying or free-tier user with some probability.

Results discussion. We first consider a scenario where each request has a 5% chance of coming from a paying user. This low percentage reflects the user distribution seen on platforms like ChatGPT, where most users belong

Dataset	Prompt length (tokens)			Decode length (tokens)		
	Median	P90	Std.	Median	P90	Std.
openchat_sharegpt4	1730	5696	2088	415	834	101

Table 5.1: Prompt and decode length (token) statistics for requests in the `openchat_sharegpt4` dataset.

to the free tier and Figures 5.2a and 5.2b present the 99th percentile TBT (P99 TBT) for paying and free-tier users, respectively. Figure 5.2c illustrates the median TTFT as a function of the request rate. In all experiments, we configure Sarathi-serve (both FCFS and SPF variants) with a token budget of 512 to ensure that the 0.1-second TBT target for paying users is met. For all SLAI variants, we use the same token budget, and set both the number of active requests and concurrent decode-phase request limit to 128. For SLAI (FCFS/SPF, fixed offset), we set the offset parameter to 10, which controls when a decode-phase request becomes time-critical, whereas for SLAI (SPF, dynamic offset), the offset is set to 5 if GPU memory utilization is below 96%, and to 10 otherwise.

TBT Behavior. Figures 5.2a and 5.2b show how SLAI dynamically prioritizes requests during their decode phase based on their TBT targets. Under Sarathi-serve, the 99th percentile TBT steadily increases for both paying and free-tier requests as the system load grows. This happens because every decode-iteration is included in every batch, and as load increases, so does the batch execution time, leading to higher delays for all requests. By contrast, SLAI handles decode-iterations differently. Requests from paying users have strict (low) TBT targets, which in most cases are always consid-

ered time-critical. As the load increases and batches take longer to run, their P99 TBT naturally increases. Free-tier requests, however, have more relaxed TBT targets. At lower loads, the scheduler can defer these decode-iterations to prioritize prefills, since they are not immediately time-critical. This initially causes their P99 TBT to rise. But as the load continues to increase and batch execution time becomes longer, the window during which a free-tier decode-iteration remains non-critical shortens. As a result, these decode-iterations become time-critical sooner and are prioritized earlier in scheduling. This leads to a drop in their P99 TBT. Eventually, at high loads, all free-tier decode-iterations are immediately marked as time-critical, and their TBT increases again—now dominated by the growing batch execution time, similar to paying users. Lastly, vLLM since it is a prefill-prioritizing scheduler ends up violating P99 TBT at a relatively low load and thus is not effective at managing decode-phase requests.

TTFT behaviour. Figure 5.2c shows the median TTFT as a function of requests per second. The vLLM policy, which prioritizes prefill-phase requests, achieves the lowest median TTFT at low loads. However, it does so by aggressively batching prefill requests at the expense of violating 99th percentile TBT latency constraints, making it unsuitable for scenarios with strict QoS requirements. Sarathi-serve improves upon this by balancing prefill and decode phases to maintain both acceptable median TTFT and TBT tail latencies. When Sarathi-serve is combined with the SPF-based policy it yields a better median TTFT than its FCFS counterpart, highlighting the benefit of

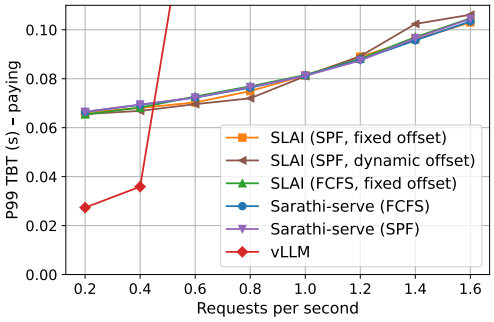
reordering prefill requests by prompt length. However, Sarathi-serve does not adapt to heterogeneous TBT deadlines across requests. SLAI (SPF, fixed offset) addresses this by selectively deferring decode-phase requests with relaxed deadlines, achieving further improvements in median TTFT. Finally, SLAI (SPF, dynamic offset) introduces dynamic decode-iteration deferral based on real-time GPU memory utilization, allowing the system to better utilize available token budget of a batch. As a result, SLAI delivers significant performance improvements: it reduces the median TTFT from 1.5 seconds (under Sarathi-Serve (FCFS)) to 0.7 seconds—a 53% improvement under high load—and increases the maximum sustainable request rate from 1.15 to 1.45 requests per second while maintaining a fixed median TTFT constraint of 0.5 seconds and meeting tail TBT latency targets, representing a 26% increase in serving capacity.

See Appendix B.1 for additional experimental results that highlight several important aspects: i) the performance of different policies as a function of prompt lengths, ii) the impact of prioritizing paying users over free-tier users during the prefill phase, and iii) how the policies compare when the proportion of paying users increases to 50% or 95%.

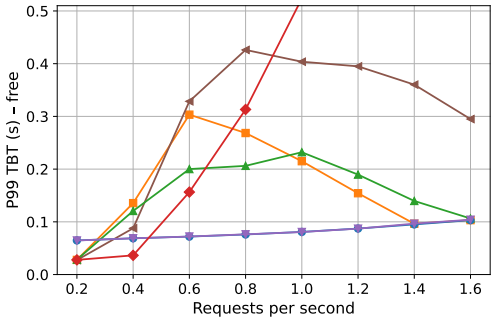
5.4 Conclusion

This paper presented a framework for designing efficient LLM inference systems. To handle heterogeneous request classes with different latency needs, we proposed the SLAI scheduler. SLAI reduces TTFT by intelligently

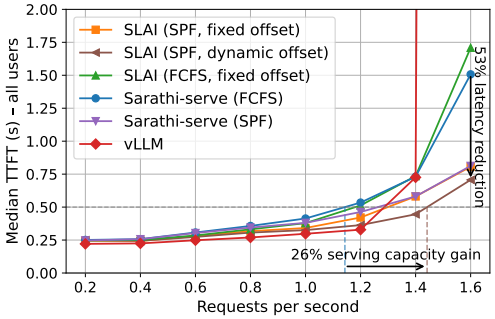
prioritizing requests while still meeting tail TBT constraints. In comparison to existing state-of-the-art, SLAI reduced the median TTFT by 53% and increased the maximum serving capacity by 26% for a fixed median TTFT, while meeting the TBT constraints.



(a) 99th percentile TBT for paying users across different request rates for a target of 0.1 seconds.



(b) 99th percentile TBT for free-tier users across different request rates for a target of 0.5 seconds.



(c) Median TTFT for all users as a function of request rate.

Figure 5.2: Performance comparison of SLAI, Sarathi-serve, and vLLM under mixed user workloads with 5% paying users. SLAI (SPF, dynamic offset) achieves the best latency-throughput trade-off.

Chapter 6

Concluding Remarks and Future Directions

6.1 Conclusion

Maximizing the system-wide utility of edge and/or cloud server resources while meeting the requirements of latency-sensitive, heterogeneous workloads with multi-resource footprints is a challenging problem. This dissertation examined two main use cases for such resources: bringing *computation* and *data caching* in proximity to clients.

Chapter 2 – Edge compute offloading. Offloading computationally heavy workloads to the edge/cloud server can serve various goals—such as saving device power, reducing average job completion time, or maximizing throughput under delay constraints. However, uncertainty in wireless channel conditions and heterogeneity in jobs and computation demands make it challenging to optimize such systems under stochastic loads. In this thesis, we showed that a measurement-based policy combining both proactive and reactive elements can address these difficulties effectively. The proactive phase, probabilistic admission control and cut assignment, performs admission control and decides which parts of a job to offload, while the reactive phase, predictive abandonment, abandons jobs that are unlikely to meet their deadlines.

Together, these components adaptively manage resources and can approach near-optimal performance in some settings.

Chapters 3 and 4 – Exploiting edge caching resources. Next, we explored how to improve cache hit rates for data object requests by focusing on two key aspects: the way data objects are represented, and how to leverage patterns in client requests when they exist.

First, we analytically demonstrated how different data object representations—especially layered representations—can influence cache performance under LRU, without the need for extensive simulations. Second, we developed a measurement-based caching policy, Least Following and Recently Used (LFRU), that infers correlations in client requests and exploits them to outperform traditional policies such as LRU and LFU across a wide range of settings.

To evaluate our approach, we created a synthetic dataset with diverse request patterns, including both uncorrelated and correlated behaviors. Our experiments showed that under structured or correlated request patterns, measurement-based policies can significantly improve performance. However, in cases where such structure is absent, traditional policies may still remain effective. Overall, our findings highlight the importance of adapting caching strategies both to the structure of client requests and to the form in which data is or can be stored.

Chapter 5 – Serving jobs that need both compute and memory resources. Lastly, we investigated how to efficiently execute LLM inference requests on a GPU, with the goal of reducing the Time To First Token (TTFT) during the prefill phase while meeting Time Between Tokens (TBT) constraints during the decode phase. These two phases place different demands on a GPU—prefill is compute-intensive, while decode is more sensitive to memory bandwidth and memory. Adding to the complexity, number of decode iterations are not known in advance, and the latency objectives for the two phases often conflict. This makes it challenging to mix and schedule such requests efficiently.

In this thesis, we proposed a measurement-based policy that dynamically prioritizes decode requests based on observed system conditions. This allows the scheduler to better allocate resources as the load changes, outperforming static policies—especially in settings with heterogeneous TBT targets. We demonstrated the effectiveness of our approach through experiments on an NVIDIA RTX ADA 6000 GPU, showing reduced TTFT compared to the state-of-the-art while meeting TBT goals for a range of loads.

These results show that simple, measurement-based
rules—when aligned with the data and the load—can keep modern edge systems both fast and efficient.

6.2 Future Research Directions

The findings in this dissertation were obtained under certain modeling assumptions that highlight the core ideas and allow careful theoretical analysis. However, several open directions remain for exploration as elaborated below:

1. Beyond Linear DAG Models for Job Offloading. The MEC offloading framework presented in this thesis assumed that jobs could be modeled as linear directed acyclic graphs, which simplifies partitioning of jobs. However, many real-world applications involve more complex workflows—e.g., tree-structured or branching graphs, or even iterative loops in model pipelines. These richer structures may require more sophisticated partitioning strategies and dynamic coordination between edge and device compute, which presents both modeling and algorithmic challenges.

2. Self-Tuning Policies. Several of the policies developed in this thesis rely on parameters tuned offline through simulation—for example, the number of requests used to infer following behavior under LFRU or the token budget and decode offset in the LLM scheduler. In practical deployments, such tuning is infeasible. An important research direction is to develop online, measurement-driven methods that enable these systems to *self-tune* in response to dynamic loads, possibly using techniques from adaptive control or reinforcement learning.

3. Routing and Caching for LLM Inference. While our work focused on intra-GPU scheduling under heterogeneous TBT constraints, future systems for LLM inference must also tackle the joint design of routing, scheduling and caching policies. For example, a router must decide whether to steer a request to a lightly loaded GPU with a cold cache, or to a busy GPU that holds relevant Key–Value (KV) tensors for the request. Similarly, cache eviction strategies must take into account both model access patterns and routing behavior. Coordinated algorithms that integrate all three dimensions—*when* to run prefill and decode phases of a request, *where* to run it, and *how* to manage the ever-growing KV cache that the model produces—are still needed.

Appendices

Appendix A

Proof for working set approximation

In this section we provide the proof our theorem for hit probability of a data object and layer.

Lemma A.1. *Under the working set memory management in the independent reference model with D data objects and V layers, the variance in the size of working set is bounded above by $\left(\frac{D \cdot V}{4} + D \cdot V \cdot (V-1)\right) \cdot \left(\delta_{\max}^{(D)}\right)^2$, where $\delta_{\max}^{(D)} = \max_{d \in \mathcal{D}, l \in \{1,2,\cdots,V\}} \delta^{(D)}(d,l)$*

Proof. Let $X^{(D)}(d,l)$ be a random variable which is 1 if data object d and layer l is in the cache at time t, and 0 otherwise. The working set at time t is given by:

$$S^{(D)}(t) = \sum_{d=1}^{D} \sum_{l=1}^{V} \delta^{(D)}(d,l) X^{(D)}(d,l), \tag{A.1}$$

then the variance $\mathrm{V}\left(S^{(D)}(t)\right)$ in the size of working set is

$$\begin{aligned}\mathrm{V}\left(S^{(D)}(t)\right) &= \mathrm{V}\left(\sum_{d=1}^{D}\sum_{l=1}^{V}\delta^{(D)}(d,l)X^{(D)}(d,l)\right)\\ &= \sum_{d=1}^{D}\sum_{l=1}^{V}\left(\delta^{(D)}(d,l)\right)^2\mathrm{V}\left(X^{(D)}(d,l)\right)+\\ &\quad 2\sum_{1\le i<d\le D}\sum_{l=1}^{V}\left(\delta^{(D)}(i,l)\delta^{(D)}(d,l)\right)\mathrm{Cov}\left(X^{(D)}(i,l),X^{(D)}(d,l)\right)+\\ &\quad 2\sum_{d=1}^{D}\sum_{1\le l<k\le V}\left(\delta^{(D)}(d,l)\delta^{(D)}(d,k)\right)\mathrm{Cov}\left(X^{(D)}(d,l),X^{(D)}(d,k)\right)\end{aligned}$$

where Cov is the covariance. Since

$$\begin{aligned}\mathrm{V}\left(X^{(D)}(d,l)\right) &\le 1/4,\\ \mathrm{Cov}\left(X^{(D)}(i,l),X^{(D)}(d,l)\right) &\le 0, i\neq d\\ \mathrm{Cov}\left(X^{(D)}(d,l),X^{(D)}(d,k)\right) &\le 1, l\neq k,\end{aligned}$$

We find that

$$\mathrm{V}\left(S^{(D)}(t)\right) \le \left(\frac{D\cdot V}{4}+D\cdot V\cdot(V-1)\right)\cdot\left(\delta_{\max}^{(D)}\right)^2 \tag{A.2}$$

□

A.0.1 Proof of 3.1

We first show that

$$\lim_{D\to\infty}\mathbb{E}\left[\frac{S^{(D)}(D\tau)}{D}\right] = \int_0^1\sum_{l=1}^{V}\Delta(x,l)dx - \int_0^1\sum_{l=1}^{V}\Delta(x,l)e^{-\tau F'(x)\sum_{v=l}^{V}g(v;x)}dx \tag{A.3}$$

where

$$\mathbb{E}\left[\frac{S^{(D)}(D\tau)}{D}\right] = \frac{1}{D}\sum_{d=1}^{D}\sum_{l=1}^{V}\delta^{(D)}(d,l)\left(1-(1-p^{(D)}(d,l))^{(D\tau-1)}\right) \tag{A.4}$$

By the Mean Value Theorem,

$$q^{(D)}(d,v) = (F'(\psi(d))/D)\cdot g(v;d/D) \tag{A.5}$$

for some $\psi(d)$ with $((d-1)/D) \leq \psi(d) \leq (d/D)$ and
$p^{(D)}(d,l) = (F'(\psi(d))/D)\sum_{v=l}^{V} q^{(D)}(d,v)$

We now use Lemma 10 from [15], which states that for each closed bounded set C,

$$(1-(c/n))^{\tau_0 n} \to e^{-\tau_0 c} \quad \text{as } n\to\infty, \text{ uniformly over all } c \text{ in } C. \tag{A.6}$$

The above is just using point wise limits. Thus, for $D \gg 1$ if

$$\left|\left(1-\frac{F'(\psi(d))}{D}\sum_{v=l}^{V} q^{(D)}(d,v)\right)^{D\tau_0} - e^{-\tau_0 F'(\psi(d))\sum_{v=l}^{V} q^{(D)}(d,v)}\right| < \epsilon \tag{A.7}$$

then one can easily show the following using the same arguments as from [15]

$$\left|\frac{1}{D}\sum_{d=1}^{D}\sum_{l=1}^{V}\delta^{(D)}(d,l)\left((1-p^{(D)}(d,l))^{(D\tau_0-1)} - e^{-\tau_0 F'(\psi(d))\sum_{v=l}^{V} q^{(D)}(d,v)}\right)\right| < \epsilon \tag{A.8}$$

Now by the definition of Riemann Integral,

$$\frac{1}{D}\sum_{d=1}^{D}\sum_{l=1}^{V}\delta^{(D)}(d,l)\left(1-e^{-\tau_0 F'(\psi(d))\sum_{v=l}^{V} q^{(D)}(d,v)}\right) \tag{A.9}$$

is an approximation to the following integral

$$\int_0^1 \sum_{l=1}^{V} \Delta(x,l)dx - \int_0^1 \sum_{l=1}^{V} \Delta(x,l) e^{-\tau F'(x) \sum_{v=l}^{V} g(v;x)} dx \tag{A.10}$$

where Riemann integrable Δ satisfies $\Delta(d/D, l) = \delta^{(D)}(d,l)$ for all D, d and l. The absolute error between Eq. A.9 and Eq. A.10 can be made smaller than ϵ for sufficiently large D. Thus, we show the result in Eq. A.3.

Additionally, using Lemma A.1, we obtain the following

$$\lim_{D\to\infty} \mathrm{V}\left(\frac{S^{(D)}(D\tau)}{D}\right) \to 0. \tag{A.11}$$

Let τ^* denote a unique solution to

$$b = \lim_{D\to\infty} \mathbb{E}\left[\frac{S^{(D)}(D\tau)}{D}\right] = \int_0^1 \sum_{l=1}^{V} \Delta(x,l)dx \quad - \\ \int_0^1 \sum_{l=1}^{V} \Delta(x,l) e^{-\tau F'(x) \sum_{v=l}^{V} g(v;x)} dx \tag{A.12}$$

For finite $D \gg 1$, this equation is approximated by

$$B = \mathbb{E}\left[S^{(D,V)}(t)\right] = \sum_{d=1}^{D}\sum_{l=1}^{V} \delta^{(D)}(d,l)\left(1 - (1 - p^{(D)}(d,l))^{(D\tau - 1)}\right) \tag{A.13}$$

with $t^* = D\tau^*$ as the unique solution for the above equation when $B = Db$.

Note that as $D \to \infty$,

$$\frac{S^{(D)}_{-(d,l)}(D\tau)}{D} \sim \frac{S^{(D)}(D\tau)}{D}$$

So,

$$\lim_{D\to\infty} \mathbb{P}\left(S^{(D)}_{-(d,l)}(D\tau) \geq B\right) = \lim_{D\to\infty} \mathbb{P}\left(S^{(D)}(D\tau)/D \geq b\right) \\ = u(\tau - \tau^*)$$

By Palm's theorem [4], the stationary LRU miss probability for data object d and layer l is

$$\begin{aligned} 1 - h^{(D)}(d,l) &= \mathbb{P}\left(S^{(D)}_{-(d,l)}(T^{(D)(d,l)}_n) \geq B\right) \\ &= \sum_{t=1}^{\infty} \mathbb{P}\left(S^{(D)}_{-(d,l)}(t) \geq B\right) p^{(D)}(d,l)(1-p^{(D)}(d,l))^{t-1} \end{aligned}$$

For $t = D\tau$, $B = Db$, $D \gg 1$, we can obtain the following with τ^* as the unique solution of Eq. A.12

$$\begin{aligned} 1 - h^{(D)}(d,l) &= \sum_{\tau=1/(D)}^{\infty} u(\tau-\tau^*) p^{(D)}(d,l)(1-p^{(D)}(d,l))^{D\tau-1} \\ &= (1-p^{(D)}(d,l))^{D\tau^*-1} \\ &= (1-p^{(D)}(d,l))^{t^*-1} \end{aligned}$$

for all data objects d and layer l. As $D \to \infty$, using Lemma 10 from [15] or point wise limits for right hand side, we obtain

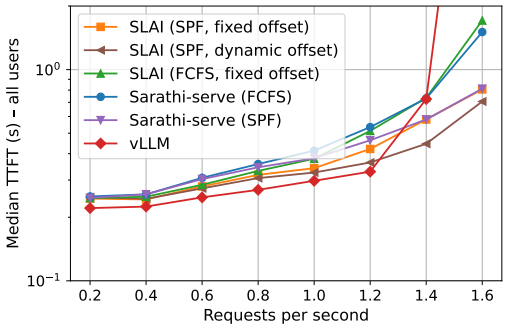
$$1 - h(d,l) = e^{-\tau^* F'(d) \sum_{v=l}^{V} g(v;d)}. \tag{A.14}$$

Appendix B

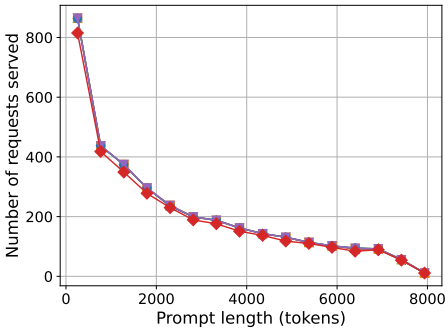
Additional experimental results for LLM inference scheduling

B.1 Results for the scenario of 5% split

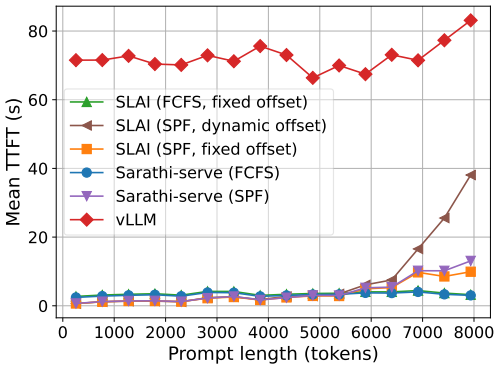
Figure B.1 presents a comparative evaluation of scheduling policies under heterogeneous TTFT and TBT constraints, with a workload comprising 5% paying users. Figure B.1a shows the median TTFT for all requests as a function of request rate, plotted on a log-scaled y-axis to highlight differences at low load. This view reveals how various schedulers handle contention-free versus saturated conditions. Figure B.1b reports the number of requests completed at the peak load of 1.6 requests per second, bucketed by prompt length. Notably, SLAI (SPF, dynamic offset) serves nearly as many requests as Sarathi-serve while achieving substantially lower median TTFT, whereas vLLM exhibits instability and fails to maintain throughput under high load. Finally, Figure B.1c plots the mean TTFT at 1.6 requests per second as a function of prompt length. Despite favoring shorter prompts, SPF-based schedulers yield a lower overall TTFT compared to FCFS variants, demonstrating the benefit of prioritizing short requests even in the presence of heterogeneous job sizes.



(a) Median TTFT for all users as a function of request rate, shown on a log-scaled y-axis to illustrate the gap at lower request rates.

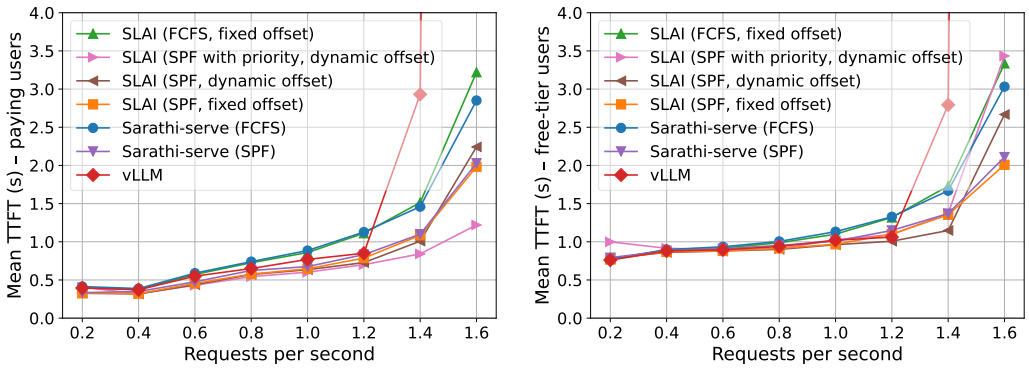


(b) Requests served at high load (1.6 req/s) versus prompt length.



(c) Mean TTFT at the high load (1.6 req/s) versus prompt length.

Figure B.1: Performance comparison of different policies under mixed user workloads with 5% paying users.



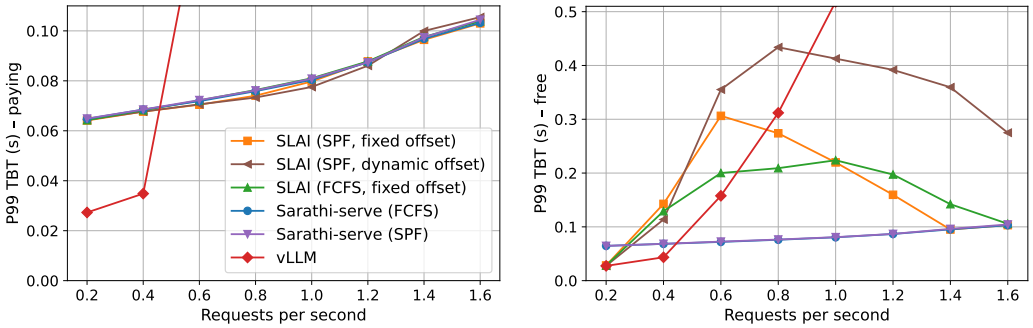
(a) Mean TTFT for paying users as a function of requests per second.

(b) Mean TTFT for free-tier users as a function of requests per second.

Figure B.2: Performance comparison of different policies under mixed user workloads with 5% paying users.

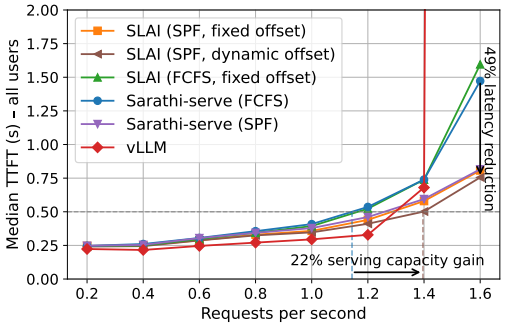
B.2 Prioritizing prefill-phase requests of paying users over free-tier users

In this section, we evaluate an additional policy: SLAI (SPF with priority, dynamic offset). This policy gives strict priority to prefill-phase requests from paying users over those from free-tier users, regardless of prompt length. In other words, it always schedules a paying user's request first. All other parameters are the same as in SLAI (SPF, dynamic offset). Figure B.2 compares this priority-based policy with other scheduling strategies. At high load (1.6 requests per second), we observe that the mean TTFT for paying users is lower than that for free-tier users. However, the improvement in TTFT for paying users is relatively small.



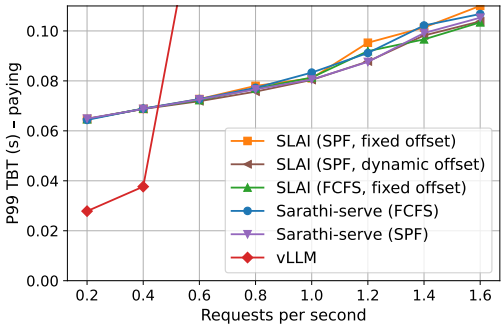
(a) 99th percentile TBT for paying users across different request rates for a target of 0.1 seconds.

(b) 99th percentile TBT for free-tier users across different request rates for a target of 0.5 seconds.

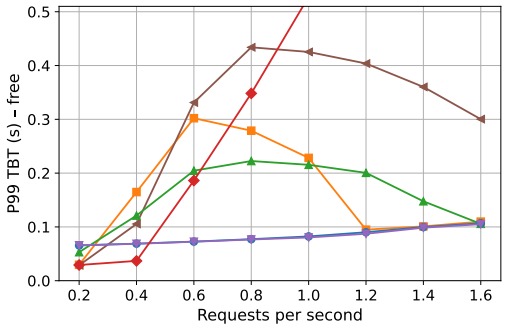


(c) Median TTFT for all users as a function of request rate. SLAI (SPF, dynamic offset) reduces TTFT from 1.5 seconds (under Sarathi-serve (FCFS)) to 0.73 seconds, and increases peak serving capacity from 1.15 to 1.4 requests per second subject to latency constraints.

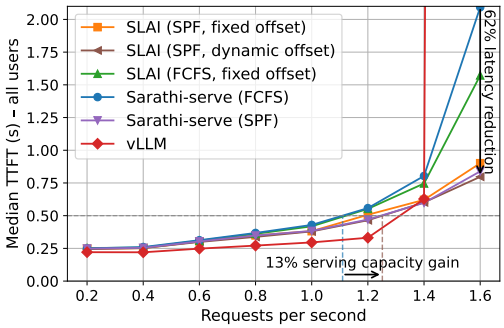
Figure B.3: Performance comparison of SLAI, Sarathi-serve, and vLLM under mixed user workloads with 50% paying users. SLAI (SPF, dynamic offset) achieves the best latency-throughput trade-off.



(a) 99th percentile TBT for paying users across different request rates for a target of 0.1 seconds.



(b) 99th percentile TBT for free-tier users across different request rates for a target of 0.5 seconds.



(c) Median TTFT for all users as a function of request rate. SLAI (SPF, dynamic offset) reduces TTFT from 2 seconds (under Sarathi-serve (FCFS)) to 0.75 seconds, and increases peak serving capacity from 1.15 to 1.25 requests per second subject to latency constraints.

Figure B.4: Performance comparison of SLAI, Sarathi-serve, and vLLM under mixed user workloads with 95% paying users. SLAI (SPF, dynamic offset) achieves the best latency-throughput trade-off.

B.3 Results for the scenario of 50% and 95% split

In this section, we present additional results for scenarios with less heterogeneity in user workloads. Figures B.3 and B.4 show results similar to those discussed earlier, but for cases where the percentage of paying users is 50% and 95%, respectively. As the proportion of paying users increases, the improvement in serving capacity under SLAI (SPF, dynamic offset) compared to Sarathi-serve (FCFS) becomes smaller. This is because a larger share of traffic now has stricter TBT constraints, leaving fewer opportunities for SLAI to defer decode-phase requests dynamically.

Bibliography

[1] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 117–134. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, p. 80–93, jan 1971. [Online]. Available: https://doi.org/10.1145/321623.321632

[3] P. A. Apostolopoulos, E. E. Tsiropoulou, and S. Papavassiliou, "Risk-aware data offloading in multi-server multi-access edge computing environment," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1405–1418, 2020.

[4] F. Baccelli and P. Brémaud, *Elements of Queueing Theory: Palm Martingale Calculus and Stochastic Recurrences, 2nd Ed.* Berlin: Springer-Verlag, 2003.

[5] F. Baccelli and B. Blaszczyszyn, *Stochastic Geometry and Wireless Networks: Volume I Theory.*, 2010.

[6] A. Bari, G. de Veciana, K. Johnsson, and A. Pyattaev, "Managing edge offloading for stochastic workloads with deadlines," in *Proceedings of the Int'l ACM Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWiM '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 99–108. [Online]. Available: https://doi.org/10.1145/3616388.3617515

[7] A. Bari, G. de Veciana, and G. Kesidis, "Fundamentals of caching layered data objects," 2025. [Online]. Available: https://arxiv.org/abs/2504.01104

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[9] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: modeling, design and experimental results," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[10] J. Chen, Y. Yang, C. Wang *et al.*, "Multi-task offloading strategy optimization based on directed acyclic graphs for edge computing," *IEEE Internet of Things Journal*, pp. 1–1, 2021.

[11] A. Dan and D. Towsley, "An approximate analysis of the lru and fifo buffer replacement schemes," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '90. New York, NY, USA: Association

for Computing Machinery, 1990, p. 143–152. [Online]. Available: https://doi.org/10.1145/98457.98525

[12] P. J. Denning and S. C. Schwartz, "Properties of the working-set model," *Commun. ACM*, vol. 15, no. 3, p. 191–198, mar 1972. [Online]. Available: https://doi.org/10.1145/361268.361281

[13] T. Q. Dinh, J. Tang, Q. D. La *et al.*, "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Transactions on Communications*, vol. 65, no. 8, pp. 3571–3584, 2017.

[14] J. Du, L. Zhao, J. Feng, and X. Chu, "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee," *IEEE Transactions on Communications*, vol. 66, no. 4, pp. 1594–1608, 2018.

[15] R. Fagin, "Asymptotic miss ratios over independent references," *Journal Computer and System Sciences*, vol. 14, no. 2, pp. 222–250, 1977.

[16] J. Famaey, S. Latré, N. Bouten, W. Van de Meerssche, B. De Vleeschauwer, W. Van Leekwijck, and F. De Turck, "On the merits of svc-based http adaptive streaming," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, 2013, pp. 419–426.

[17] Q. Fan, X. Li, J. Li, Q. He, K. Wang, and J. Wen, "Pa-cache: Evolving learning-based popularity- aware content caching in edge networks," *IEEE*

Transactions on Network and Service Management, vol. 18, no. 2, pp. 1746–1757, 2021.

[18] F. Faticanti and G. Neglia, "Optimistic online caching for batched requests," 2023. [Online]. Available: https://arxiv.org/abs/2310.01309

[19] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "Competitive paging algorithms," *Journal of Algorithms*, vol. 12, no. 4, pp. 685–699, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/019667749190041V

[20] C. Fricker, P. Robert, and J. Roberts, "A Versatile and Accurate Approximation for LRU Cache Performance," in *Proc. International Teletraffic Congress*, Krakow, Poland, 2012.

[21] F. Hartanto, J. Kangasharju, M. Reisslein, and K. Ross, "Caching video objects: Layers vs versions?" *Multimedia Tools Appl.*, vol. 31, no. 2, p. 221–245, nov 2006. [Online]. Available: https://doi.org/10.1007/s11042-006-0037-z

[22] G. Hasslinger, M. Okhovatzadeh, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "An overview of analysis methods and evaluation results for caching strategies," *Computer Networks*, vol. 228, p. 109583, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128623000282

[23] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," 2024. [Online]. Available: https://arxiv.org/abs/2401.08671

[24] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, Y. Dong, and Y. Wang, "Flashdecoding++: Faster large language model inference on gpus," 2024. [Online]. Available: https://arxiv.org/abs/2311.01282

[25] H. Hoppe, "Progressive meshes," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 99–108. [Online]. Available: https://doi.org/10.1145/237170.237216

[26] Y. Im, P. Prahladan, T. H. Kim, Y. G. Hong, and S. Ha, "Snn-cache: A practical machine learning-based caching system utilizing the inter-relationships of requests," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, 2018, pp. 1–6.

[27] K. Jain, A. Parayil, A. Mallick, E. Choukse, X. Qin, J. Zhang, Íñigo Goiri, R. Wang, C. Bansal, V. Rühle, A. Kulkarni, S. Kofsky, and S. Rajmohan, "Intelligent router for llm workloads: Improving performance through workload-aware load balancing," 2025. [Online]. Available: https://arxiv.org/abs/2408.13510

[28] P. R. Jelenković and A. Radovanović, "The persistent-access-caching algorithm," *Random Struct. Algorithms*, vol. 33, no. 2, p. 219–251, sep 2008.

[29] P. R. Jelenković, "Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities," *The Annals of Applied Probability*, vol. 9, no. 2, pp. 430–464, 1999. [Online]. Available: http://www.jstor.org/stable/2667340

[30] P. R. Jelenković and A. Radovanović, "Asymptotic optimality of the static frequency caching in the presence of correlated requests," *Operations Research Letters*, vol. 37, no. 5, pp. 307–311, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167637709000510

[31] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon, "Computation offloading for machine learning web apps in the edge server environment," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1492–1499.

[32] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[33] B. Jiang and Y. Mu, "Russian doll network: Learning nested networks for sample-adaptive dynamic inference," in *2021 IEEE/CVF International*

Conference on Computer Vision Workshops (ICCVW), 2021, pp. 336–344.

[34] J. Juravsky, B. Brown, R. Ehrlich, D. Y. Fu, C. Ré, and A. Mirhoseini, "Hydragen: High-throughput llm inference with shared prefixes," 2024. [Online]. Available: https://arxiv.org/abs/2402.05099

[35] A. K. Kamath, R. Prabhu, J. Mohan, S. Peter, R. Ramjee, and A. Panwar, "Pod-attention: Unlocking full prefill-decode overlap for faster llm inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. ACM, Mar. 2025, p. 897–912. [Online]. Available: http://dx.doi.org/10.1145/3676641.3715996

[36] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ASPLOS*, vol. 52, no. 4, p. 615–629, 2017.

[37] E. Kim, C. Ahn, and S. Oh, "Nestednet: Learning nested sparse structures in deep neural networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8669–8678.

[38] W. King, "Analysis of paging algorithms," *Proc. IFIP Congress*, pp. 485–490, 1971.

[39] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman, "Rl-cache: Learning-based cache admission for content delivery," *IEEE Journal on*

Selected Areas in Communications, vol. 38, no. 10, pp. 2372–2385, 2020.

[40] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," in *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2018, pp. 1–6.

[41] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," 2023. [Online]. Available: https://arxiv.org/abs/2309.06180

[42] L. Lai, F.-C. Zheng, W. Wen, J. Luo, and G. Li, "Dynamic content caching based on actor-critic reinforcement learning for iot systems," in *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*, 2022, pp. 1–6.

[43] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," 2023. [Online]. Available: https://arxiv.org/abs/2211.17192

[44] J. Liu, J. Ren, Y. Zhang, X. Peng, Y. Zhang, and Y. Yang, "Efficient dependent task offloading for multiple applications in mec-cloud system," *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2147–2162, 2023.

[45] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, M. Maire, H. Hoffmann, A. Holtzman, and J. Jiang, "Cachegen: Kv cache compression and streaming for fast large language model serving," 2024. [Online]. Available: https://arxiv.org/abs/2310.07240

[46] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021.

[47] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *J. ACM*, vol. 68, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3447579

[48] X. Lyu, H. Tian, W. Ni, Y. Zhang *et al.*, "Energy-efficient admission of delay-sensitive tasks for mobile edge computing," *IEEE Transactions on Communications*, vol. 66, no. 6, pp. 2603–2616, 2018.

[49] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, ser. NetAI'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 48–53. [Online]. Available: https://doi.org/10.1145/3229543.3229555

[50] ——, "Making content caching policies 'smart' using the deepcache framework," vol. 48, no. 5. New York, NY, USA: Association for

Computing Machinery, Jan. 2019, p. 64–69. [Online]. Available: https://doi.org/10.1145/3310165.3310174

[51] NVIDIA, "Fastertransformer," 2025. [Online]. Available: https://github.com/NVIDIA/FasterTransformer

[52] NVIDIA, "TensorRT-LLM: A tensorrt toolbox for optimized large-language-model inference," 2025. [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM

[53] OpenAI, "Chatgpt," 2025. [Online]. Available: https://chat.openai.com

[54] A. Ortega, F. Carignano, S. Ayer, and M. Vetterli, "Soft caching: web cache management techniques for images," in *Proc. First Signal Processing Society Workshop on Multimedia Signal Processing*, 1997, pp. 475–480.

[55] G. Papaioannou and I. Koutsopoulos, "Tile-based caching optimization for 360° videos," in *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. Mobihoc '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 171–180. [Online]. Available: https://doi.org/10.1145/3323679.3326515

[56] T. Shanableh and M. Ghanbari, "Heterogeneous video transcoding to lower spatio-temporal resolutions and different encoding formats," *IEEE Transactions on Multimedia*, vol. 2, no. 2, pp. 101–110, 2000.

[57] W. Shi, Y. Hou, S. Zhou, Z. Niu, Y. Zhang, and L. Geng, "Improving device-edge cooperative inference of deep learning via 2-step pruning," in *IEEE Conference on Computer Communications Workshops*, 2019, pp. 1–6.

[58] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, p. 202–208, feb 1985. [Online]. Available: https://doi.org/10.1145/2786.2793

[59] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 173–191. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-biao

[60] C. Wang, C. Liang, F. R. Yu *et al.*, "Computation offloading and resource allocation in wireless cellular networks with mobile edge computing," *IEEE Transactions on Wireless Communications*, vol. 16, no. 8, pp. 4924–4938, 2017.

[61] G. Wang, S. Cheng, X. Zhan, X. Li, S. Song, and Y. Liu, "Openchat: Advancing open-source language models with mixed-quality data," 2024. [Online]. Available: https://arxiv.org/abs/2309.11235

[62] J. Wang, J. Hu, G. Min *et al.*, "Fast adaptive task offloading in edge

computing based on meta reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2021.

[63] A. R. Ward and W. Whitt, "Predicting response times in processor-sharing queues," in *In Proc. of the Fields Institute Conf. on Comm. Networks*, 2000, pp. 1–29.

[64] M. Xu, F. Qian, M. Zhu *et al.*, "Deepwear: Adaptive local offloading for on-wearable deep learning," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 314–330, 2020.

[65] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, "Flashinfer: Efficient and customizable attention engine for llm inference serving," 2025. [Online]. Available: https://arxiv.org/abs/2501.01005

[66] C. You, K. Huang, H. Chae *et al.*, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Transactions on Wireless Communications*, vol. 16, no. 3, pp. 1397–1411, 2017.

[67] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[68] T. Zhang, "Data offloading in mobile edge computing: A coalition and pricing based approach," *IEEE Access*, vol. 6, pp. 2760–2767, 2018.

[69] W. Zhang and Y. Wen, "Energy-efficient task execution for application as a general topology in mobile cloud computing," *IEEE Transactions on Cloud Computing*, vol. 6, no. 3, pp. 708–719, 2018.

[70] W. Zhang, Y. Wen, and D. O. Wu, "Collaborative task execution in mobile cloud computing under a stochastic wireless channel," *IEEE Transactions on Wireless Communications*, vol. 14, no. 1, pp. 81–93, 2015.

[71] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, "SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1229–1246. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

[72] Y. Zhao, S. Yang, K. Zhu, L. Zheng, B. Kasikci, Y. Zhou, J. Xing, and I. Stoica, "Blendserve: Optimizing offline inference for auto-regressive large models with resource-aware batching," *arXiv preprint arXiv:2411.16102*, 2024.

[73] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," 2024. [Online]. Available: https://arxiv.org/abs/2312.07104

[74] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "{DistServe}: Disaggregating prefill and decoding for goodput-optimized

large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 193–210.

[75] C. Zhu, G. Pastor, Y. Xiao, and et.al., "Fog following me: Latency and quality balanced task allocation in vehicular fog computing," in *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2018, pp. 1–9.

Vita

Agrim Bari is pursuing his Ph.D. degree in electrical and computer engineering at the University of Texas at Austin (UT Austin), USA, under the supervision of Dr. Gustavo de Veciana. He received his B.Tech. degree in electrical engineering from Indian Institute of Technology Kanpur in 2019 and M.S. degree in electrical and computer engineering from UT Austin in 2023. He was an intern at Qualcomm Wireless R&D, Bridgewater, NJ in the summer of 2022, and Qualcomm Wireless R&D, San Diego, CA in the summer of 2023. His research interest lies in building modern systems; he has worked on projects spanning 6G Wireless Networks, caching for Virtual Reality, and LLM inference systems optimization.

Permanent address: agrim.bari@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.