Performance Modeling of Distributed Data Processing in Microservice Applications

YICHENG GAO and GIULIANO CASALE, Imperial College London, UK
REKHA SINGHAL, Tata Consultancy Services, USA

Microservice applications are increasingly adopted in distributed data processing systems, such as in mobile edge computing and data mesh architectures. However, existing performance models of such systems fall short in providing comprehensive insights into the intricate interplay between data placement and data processing. To address these issues, this paper proposes a novel class of performance models that enables joint analysis of data storage access workflows, caching, and queueing contention. Our proposed models introduce a notion of access path for data items to model hierarchical data locality constraints. We develop analytical solutions to efficiently approximate the performance metrics of these models under different data caching policies, finding in particular conditions under which the underlying Markov chain admits a product-form solution. Extensive trace-driven simulations based on real-world datasets indicate that service and data placement policies based on our proposed models can respectively improve improve the average response time by up to 35% and 37% in edge and data mesh case studies compared to baseline resource allocation heuristics.

Additional Key Words and Phrases: microservice, performance modeling, data placement, data caching, layered network.

ACM Reference Format:
Yicheng Gao, Giuliano Casale, and Rekha Singhal. 2024. Performance Modeling of Distributed Data Processing in Microservice Applications. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 1, 1, Article 1 (January 2024), 29 pages. https://doi.org/10.1145/3687265

1 Introduction

Recent trends in software and systems engineering have promoted the decomposition of monolithic applications into distributed architectures consisting of a collection of self-contained microservices. Each self-contained microservice can be independently developed and deployed in containers or virtual machines and interacts with the other microservices through lightweight mechanisms such as RESTful APIs [**?** **?**]. Compared to monolithic systems, a microservices architecture enables the dynamic deployment, migration, and removal of each service during runtime, enhancing flexibility, scalability, and adaptability [**?** **?**], thus prompting their rapid adoption in industry [**?**]. Such features facilitate distributed processing of microservice applications, which can improve resource efficiency and resilience in various applications, *e.g.,* mobile edge computing [**?**], stream processing [**?**], and data mesh architectures [**?**]. However, the complex inter-dependencies of computational processing, data placement, caching, and limited in-memory storage capacity poses a challenge in devising optimal resource management policies for such systems.

To design reliable performance predictions to drive the definitions of such policies, we consider two distinct categories of services: in-memory data storage services and computational services. In-memory data storage services, *e.g.,* Memcached [**?**], are usually modeled using stochastic caching analysis [**?**]. Instead, computational services perform different kinds of computations on the data, frequently resulting in contention among competing requests that may

Authors' addresses: Yicheng Gao; Giuliano Casale, {y.gao20,g.casale}@imperial.ac.uk, Imperial College London, London, UK; Rekha Singhal, Tata Consultancy Services, New York, USA, rekha.singhal@tcs.com.

Manuscript submitted to ACM

be modeled through stochastic queueing models [**?**]. Existing studies [**? ? ? ? ? ? ? ? ?**] mainly predict performance based on the modeling of computational services alone, which however overlooks the role data placement, caching, and access locality plays in end-to-end performance. As distributed data processing can feature complex and varying dependencies between in-memory storage and computational services [**?**], the latency interplay of queueing for service access and cache hits/misses is non-negligible, calling for novel frameworks that can jointly model in-memory storage and computational service performance.

We contribute several extensions and theoretical results that help to address these challenges. First, we generalize existing product-form solutions for list-based caches to provide an analytical formalism to describe flexible data placement. In particular, we use the notion of cache list to model an in-memory storage node and our analytical formalism is extended to describe allowed locations where a data item can be stored in a hierarchically distributed network of in-memory storage nodes. The latter is achieved by introducing a notion of data item access path, which is essential to model organizational, regulatory (*e.g.*, privacy-related), or geographical constraints on data placement. We study the exact solution of the above storage models under random-replacement (RR) and least-recently-used (LRU) caching policies, adopting a time-to-live (TTL) approximation to analyze the latter. These results extend earlier work on computational methods for caching analysis [**? ?**] to the generalized class of hierarchical caches studied in the paper.

Next, we integrate the above hierarchical caching model into the layered queueing network (LQN) formalism [**?**], enabling the analysis of data processing by means of workflow graphs that describe not only computation, but also data access dynamics and the effect of cache misses upon accessing an in-memory data storage node. The resulting class of generalized LQNs, called in the paper layered caching-queueing (LCQ) models, are analyzed by a novel fixed-point iteration method to approximate steady-state performance metrics.

Lastly, we validate the hierarchical caching model and the proposed LCQ model.[1] We have also implemented the proposed performance models within the LINE[2] tool [**?**]. Then, using this implementation, we report on extensive experiments based on real-world traces for edge and data mesh scenarios to illustrate the potential gains from adopting our models in service and data placement problems. For edge service placement, we find in particular that our proposed resource allocation method achieves a 35% improvement in response time and significantly reduces memory usage compared to baseline heuristics methods. For the data mesh scenario, our analytical solutions achieve a reduction in mean response time of up to 37%.

The rest of the paper is organized as follows. Section 2 presents related work and introduces the layered queueing network modeling formalism. Section 3 illustrates the modeling of hierarchical caches and gives the analytical methods under different replacement policies. Section 4 develops the integrated caching-queueing formalism and presents the analytical solution to the LCQ model. Section 5 validates the proposed performance models. Section 6 and Section 7 evaluate the proposed models using real-world traces for the application of edge service placement and data mesh, respectively. Finally, Section 8 concludes the paper and presents the future work. Proofs are given in the Appendix.

2 Background

In the real world, most data processing has complex dependencies between in-memory data storage and computational services. This is illustrated in Figure 1, which shows an example of a social network architecture from [**? ?**], where the client requests are first forwarded to a front-end load balancer to be routed to multiple microservices. Subsequently, these requests are processed by a series of logic mid-tiers, encompassing an array of functions, such as creating posts,

[1]The validation datasets are available at https://doi.org/10.5281/zenodo.6491327
[2]The LINE tool is available at http://line-solver.sourceforge.net/

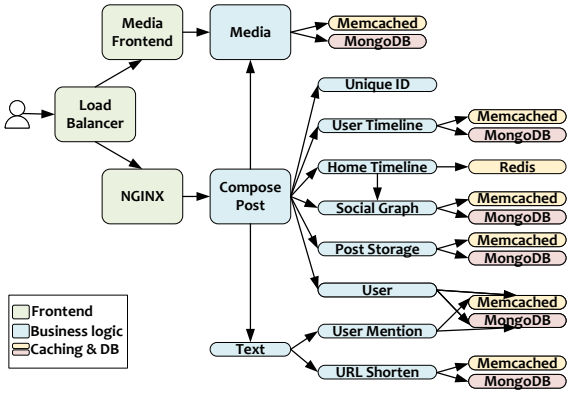


Fig. 1. A dependency graph of social network microservice architecture [? ?]

or reading timelines. Microservices-based architectures of this kind feature in-memory caching tiers, like Memcached and Redis, and persistent databases, such as MongoDB, to store critical data, such as posts, profiles, and media. This illustrates a concrete architecture where it is difficult to disentangle end-to-end performance by separating the analysis of computational and in-memory data storage services. This offers a motivation for investigating the interaction between computed-oriented and data-oriented services in more depth.

2.1 Related Work

We note in particular that existing performance modeling methods for distributed data processing of microservice applications are primarily centered on computational services for resource optimization. For example, several studies [**?** **?**] use queueing models to make performance predictions, often leveraging $M/M/c$ or $M/GI/1$ queues [**?**]. However, isolated queueing systems are insufficient to represent complex interdependencies between microservices, which can result in emerging phenomena, such as bottleneck switches [**?**].

To address this issue, many studies [**?** **?** **?** **?** **?** **?**] adopt graph-based approaches to model service dependencies. For example, GrandSLAm [**?**] builds a directed acyclic graph (DAG) of microservices from a high-level language input to calculate partial deadlines for each microservice stage. Similarly, a microservice level parallelism applying the DAG modeling of requests for better resource utilization is proposed in [**?**]. Wang *et al.*[**?**] use DAGs to model the microservices deployment problem. Fu *et al.* [**?**] utilize DAGs to model microservices and divide the graph into partitions for mapping to different nodes. In addition, Wang *et al.*[**?**] describe each workflow as a two-tuple and define a cost optimization problem with deadline constraints to optimize task scheduling of microservices. AlphaR [**?**] leverages a bipartite graph to extract the temporal characteristics of microservices.

In the above-mentioned research, there are two limitations that we try to overcome in our work. First, graph-based modeling of computational services does not encompass scheduling strategies, resource contention, and stochasticity of workloads. These aspects can be however studied with stochastic performance models, such as LQNs [**?**]. Next, current modeling approaches dedicated to computational services neglect the significant role played by in-memory storage services in performance. This is a limitation since the invocation of services such as Memcached or other in-memory storage is nowadays fairly common to reduce response times. In some large-scale applications with complex dependencies, the call ratio of Memcached can reach 50% [**?**]. On the other hand, the processing time of in-memory

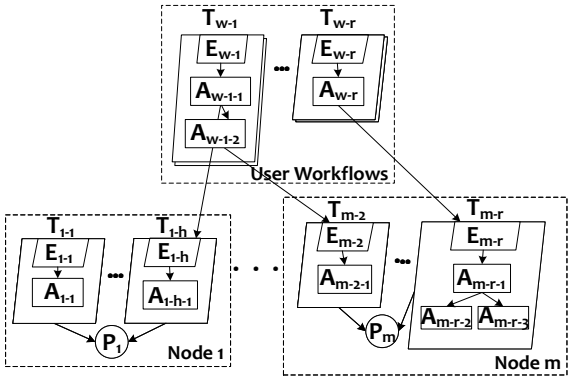


Fig. 2. An example of layered queueing networks modeling for distributed processing of microservices

storage services has a broad impact on overall efficiency. Some applications can spend up to 80% of their overall execution time reading data [**?**]. However, no layered queueing formalism today includes stochastic models of in-memory data storage services.

2.2 Layered Queueing Network Modeling

Motivated by the above state-of-the-art, we propose to establish a unified performance modeling approach for both computational and in-memory storage services. Therefore, we introduce here LQN modeling to serve as a basis to fit our purpose, which is proven to perform better in stochastic modeling of distributed workflows [**?**][3]. This allows us to further integrate the description of both types of services to enable the modeling of systems such as those shown in Figure 1.

Figure 2 summarizes an example of the LQN model describing several workflows requested by clients running on a set of m nodes, where each node provides multiple services. A distributed architecture is seen as consisting of a set of nodes in a client-server relationship. Each node m is restricted to one of the following types:

- *Processor*: The computational resources on each node m are modeled as processor P_m, shown as a circle, obeying the processor-sharing scheduling policy.
- *Task*: The co-located servers are modeled as a set of tasks T_{mc}, shown as the larger parallelograms, each representing service c running on processor P_m. Each task has a queue to model first-come first-serve (FCFS) admission control and the service time follows a phase-type distribution.
- *Entry*: Each task publishes at least one entry E_{mc}, shown as a small parallelogram, which represents the endpoint for calls to service c. That is, upon invoking an entry E_{mc}, this will lead to a chain of activities being executed on the underpinning processor.
- *Activity*: Each entry invokes a sequence of activities A_{mci}, where $i = 1, \ldots, I_c$ and I_c is the number of activities for service c. Activities are shown as a series of rectangles under the entries organized in the form of a DAG, to represent independent or dependent executions. Activities may trigger synchronous or asynchronous calls to entries, shown as arrows, resulting in complex service workflow.

[3]Full details of stochastic LQN-based modeling are given in Appendix C.

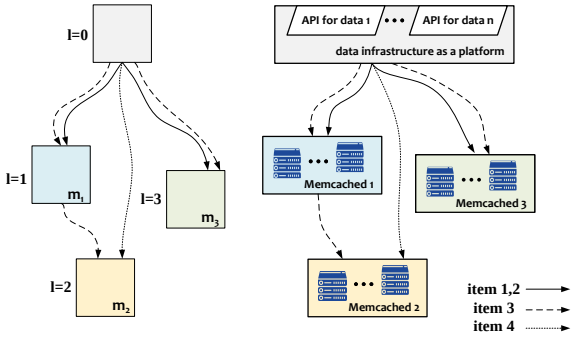


Fig. 3. An example of a tree-structured cache for data mesh. The shaded node (cache list $l = 0$) represents out-of-cache data items.

The clients are modeled in the LQN formalism as *reference tasks*. A reference task is a modeling abstraction used to describe a workload class, in our case a client workflow DAG. That is, the workflows describing the client request chains, *e.g.,* T_{w-1}, are described as DAGs within the reference task, called activity graphs in LQN terminology, to represent the precedences between the jobs. The mean queue length of the infinite server on the reference task is defined as the multiplicity of the workflow. When requested jobs are assigned to different nodes, the activity graph will feature specific activities that make synchronous calls to corresponding nodes based on their precedence, such as A_{w-1-1} and A_{w-1-2}.

The LQN models we have described above allow us to obtain analytical estimates of throughputs, response times (*i.e.,* latency per unit invocation), mean queue-lengths (*i.e.,* backlogs), and utilizations for each component type, for example using the algorithms given in [**?** **?**]. We focus on response time throughout the rest of the paper, but other reference metrics can be easily generated from the models.

3 Data Distribution Modeling

3.1 Cache Modeling

Hierarchical caching is a valuable mechanism to reduce data access latency in real-world distributed systems, particularly for enhancing data locality and faster access times [**?**]. Different from cache network [**?**], hierarchical caching can take advantage of the proximity and locality of data by following a predetermined path within the cache hierarchy. For example, in a data mesh, different domain teams may access their respective data products at a hierarchy of cache nodes, each located closer to the team than the data source. In this section, we introduce an analytical model that may be suitable to represent such scenarios.

An illustrating example is shown in Figure 3, depicting a toy caching architecture serving four data items (*i.e.,* data products). Here, the data mesh architecture on the right-hand side is abstracted, in a simplified fashion, as a tree-structured caching model, shown on the left-hand side, where different cache lists correspond to different Memcached nodes[4]. Although more sophisticated models of cache networks are possible, we argue that applying a simple abstraction such as the one in Figure 3 allows to tractably reason on data placement problems, capturing essential aspects of access locality and capacity constraints. In the following, we develop a mathematical analysis of a class of tree-structured caches considered in the paper that generalizes existing models [**?**].

[4]In the following, the terms cache list and cache node are used interchangeably.

3.1.1 Hierarchical Data Distribution In the analytical cache models developed in this paper, we consider a total of M cache nodes distributed in h layers arranged in a tree topology. Each cache node has a capacity of m_l data items, $l = 1, 2, ..., M$. The total cache capacity is $m = \sum_{l=1}^{M} m_l$ items. In total, data consists of I items accessible through requests, *e.g.,* via REST APIs. The size of each data item is denoted as s_i, $i = 1, 2, ..., I$, and the total size of data items is $n = \sum_{i=1}^{I} s_i$. Node $l = 0$ is a virtual cache node representing all remote data sources, *i.e.,* virtually storing all data items outside the cache nodes, with a total capacity of $m_0 = n - m$ items.

We describe the data distribution across the cache nodes using Markov chains, however, this implies that representing item sizes presents difficulties, since the analysis normally either loses product-form solution or becomes computationally difficult [**?**]. Hence, in what follows, we take the common assumption that the cache model abstracts a simplified scenario where items have identical sizes ($s_i = 1, \forall i$). Thus, we keep track of the presence of an item at a particular node, but not of its size. This restriction is lifted later by accounting for total available cache capacity directly in the optimization programs used to distribute data, as shown in Section 6.1 and Section 7.2.

In accordance with the independent reference model (IRM), we specify the requests to data items by the users as u independent request streams. Stream v is assumed to be a Poisson process with rate $\lambda_{vk}(l)$ for data item k stored in cache node l, where $v = 1, \ldots, u$; $k = 1, \ldots, I$; $l = 1, \ldots, h$.

3.1.2 Replacement Policies with Access Paths Cache nodes in our model can exchange data items to improve the locality of access. A data item can be stored at a single cache node at a time. Each cache node l has only one parent cache node that can exchange data items with, but a set of child nodes, denoted by sets $p(l)$ and $\boldsymbol{b}(l)$ respectively. If $\boldsymbol{b}(l) = \emptyset$, node l is a leaf node.

The movement of items across the hierarchical cache is regulated by the cache replacement policy. Multiple replacement policies exist for individual caches, *e.g.,* random replacement (RR), first in first out (FIFO), least recently used (LRU). If the cache is partitioned in nodes (*i.e.,* lists) and there is control over the probability that an item moves from a parent node to its children, the generalizations of these replacement policies are called RR-c($\boldsymbol{m}$), FIFO-c($\boldsymbol{m}$), LRU-c($\boldsymbol{m}$) [**?**].

Due to their practicality, we focus on RR-c($\boldsymbol{m}$) and LRU-c($\boldsymbol{m}$) policies and propose in this paper to generalize them to describe scenarios such as the one in Figure 3, in which items may go directly from the source (node $l = 0$) to any of the cache nodes (*e.g.,* node 2). This is not allowed in existing policies such as RR-c($\boldsymbol{m}$) and LRU-c($\boldsymbol{m}$), but it is a requirement in systems where data may be requested directly from the data sources at any location, *e.g.,* close to the team using that data product.

To resolve these issues, we introduce the concept of access path $\mathcal{P}_{vi}(j)$, *i.e.* a sequence of nodes visited by a data item i that starts outside the cache until it is promoted to node j under requests from stream v only. If data item i cannot reach node j under requests coming from stream v, we set $\mathcal{P}_{vi}(j) = \emptyset$. With the definition of access paths, the resulting variants of the above replacement policies are denoted as RR-a($\boldsymbol{m}$) and LRU-a($\boldsymbol{m}$). The two policies are defined as follows.

In both policies, if a cache hit for item k in a non-leaf node l occurs, then item k swaps its position with an item in one of its child nodes j in accordance with the following rule:

- RR-a($\boldsymbol{m}$): a child node j is chosen with probability $c_{vk}(l, j)$, $j \in \boldsymbol{b}(l)$ and an item k' within j is chosen uniformly at random for eviction. Item k then replaces item k' in node j and, vice versa, item k' is stored in place of item k in node l.
- LRU-a($\boldsymbol{m}$): similar to RR-a($\boldsymbol{m}$), except that item k' is chosen as the least recently used item in node j.

If a cache miss occurs, then item k evicts an item from node l with probability $c_{vk}(0,l)$ to replace it. Note that the ability to specify $c_{ok}(0,l)$ in our models goes beyond caches studied under RR-c($\boldsymbol{m}$) and LRU-c($\boldsymbol{m}$), where items triggering an eviction are restricted to enter the cache at node $l = 1$.

An example of access paths can be illustrated by the case presented in Figure 3 with $\boldsymbol{m} = (1,1,1)$ and $n = 4$. In the diagram, an arrow for data item i between nodes l and l' indicates that there exists a stream v such that $c_{vi}(l,l') > 0$. In-memory storage locations that are not reachable according to the access path determined by the geography-based constraints are not listed. According to our definitions, there are access paths, *e.g.*, $\mathcal{P}_{v3}(2) = \{(0,1),(1,2)\}$ and $\mathcal{P}_{v4}(2) = \{(0,2)\}$, and unreachable paths, *e.g.*, $\mathcal{P}_{v1}(2) = \mathcal{P}_{v2}(2) = \mathcal{P}_{v4}(1) = \mathcal{P}_{v4}(3) = \emptyset$.

3.2 Analytical Methods

To analyze the performance of the class of hierarchical cache models we consider, we first generalize an exact solution in [**?**] for the RR-$a(\boldsymbol{m})$ policy to enable each data item to directly enter cache nodes at different layers. Then, considering the high computational complexity of calculating large-scale LRU caches, we generalize existing time-to-live (TTL) approximations to the LRU-$a(\boldsymbol{m})$ policy, leveraging the fact that prior work has obtained asymptotically exact results for LRU($\boldsymbol{m}$) in linear caches [**?**].

A. Exact Solution for RR-$a(\boldsymbol{m})$ Policy

We first focus on RR-$a(\boldsymbol{m})$ policy and leverage continuous-time Markov chain (CTMC) to model the replacement policy. We define the state vector $\boldsymbol{s} = [s(k,j)]$ to represent the position of item k in node j in the cache. The reachable state space is $\boldsymbol{s} \in \mathcal{S}$. Thus, we set $\pi(\boldsymbol{s})$ the equilibrium probability of the model. A product-form result now holds.

Theorem 1. *Assume that if $\mathcal{P}_{vk}(j) = \emptyset$ then $\lambda_{vk}(j) = 0$ and otherwise that access paths are stream-independent, i.e., $\mathcal{P}_{vk}(j) = \cup_{v=1}^{u} \mathcal{P}_{vk}(j) = \mathcal{P}_{k}(j)$. Then, RR-$a(\boldsymbol{m})$ policy admits the equilibrium distribution*

$$\pi(\boldsymbol{s}) = \frac{\prod_{j=0}^{M} \prod_{k=1}^{m_j} \alpha_{s(k,j)j}}{E(\boldsymbol{m})} \tag{1}$$

for all recurrent states $\boldsymbol{s} \in \mathcal{S}$, where we define the normalizing constant

$$E(\boldsymbol{m}) = \sum_{\boldsymbol{s}\in\mathcal{S}} \prod_{j=0}^{M} \prod_{k=1}^{m_j} \alpha_{s(k,j)j} \tag{2}$$

and where

$$\alpha_{kj} = \begin{cases} \prod_{(l,l')\in\mathcal{P}_k(j)} \sum_{v=1}^{u} \lambda_{vk}(l) c_{vk}(l,l') & \text{if } j > 0 \\ 1 & \text{if } j = 0 \end{cases}$$

for all items $k = 1,\ldots,n$ and cache nodes $j = 0,\ldots,M$.

We refer to α_{kj} as the *cost factor* for item k to access node j. The cost factor differs from the access cost in that it accounts for both the costs and the item request rates involved in item i movement from outside the cache into node j. In this way, the normalizing constant $E(\boldsymbol{m})$ can be computed same as the methods in [**?**]. The proof is given in Appendix A, together with an illustrative example in Appendix B.

The result generalizes the prior work on tree-structured caches accounting for the access paths of items. In particular, the above result states that if the locations at which a data item can be stored are the same for all users of that data item, then it is possible to tractable analyze the caching model using a product-form solution. The performance measure for

the analytical model is denoted by the total miss ratio $p_{miss}^{(v)}$ for stream v, which can be iteratively approximated by the fixed point iteration method (FPI) [?] as:

$$p_{miss}^{(v)} = 1 - \sum_{k,j} \pi_{kj}^{(t)}, \quad k = 1, 2, ..., n; j = 1, ..., M \tag{3}$$

where t is an iteration number and $\pi_{kj}^{(t)} = \alpha_{kj}\xi_j^{(t)} \left(1 + \sum_{l=1}^{h} \alpha_{kl}\xi_l^{(t)}\right)^{-1}$ represents the probability of item k residing in list j at equilibrium and $\xi_j^{(t)} = m_j \left(\sum_{k=1}^{n} \alpha_{kj}(1 - \sum_{j=1}^{h} \pi_{kj}^{(t-1)})\right)^{-1}$. Initially, FPI sets $\pi_{kj}^{(0)} = 0$ for all items k if $j > 0$, and $\pi_{k0}^{(0)} = 1$ otherwise. The iteration continues until either the miss ratio $p_{miss}^{(v)}$ has converged or the maximum number of iterations has been reached. The corresponding hit ratio for stream v is given by $p_{hit}^{(v)} = \lambda_v - p_{miss}^{(v)}$.

B. TTL Approximation for LRU-$a(\boldsymbol{m})$ Policy

We now focus on the LRU-$a(\boldsymbol{m})$ replacement policy. Such caches are difficult to analyze even in the basic LRU($\boldsymbol{m}$) setting, for which prior work has used TTL approximations [?]. The TTL approximation assumes an infinite capacity for each node to approximate the LRU($\boldsymbol{m}$) policy, thus evicting the data items based on the expiration time rather than the usage history. In this section, we illustrate how to extend the formulas in [?] to encompass access paths, thus generalizing the approximation to the LRU-$a(\boldsymbol{m})$ policy, *i.e.,* to non-uniform access costs and access paths.

Under TTL approximation, each cache node l stores data items for a deterministic time period of duration T_l, *i.e.,* the time-to-live value. If a data item k is not accessed until the expiration of its time to live, the item moves from node l to its parent node $p(l)$ and the corresponding cost factor is

$$\alpha_k(l, p(l)) = e^{-\sum_{v=1}^{u} \lambda_{vk}(l) T_l}, \qquad l = 1, ..., M \tag{4}$$

If data item k is requested before exceeding T_l in non-leaf node l, it is promoted to its child node j with probability $c_{vk}(l, j)$. Otherwise, it keeps itself in leaf node j and resets its time-to-live. The cost factor of data item k moving from current node l to node j is represented as

$$\alpha_k(l, j) = 1 - e^{-\sum_{v=1}^{u} c_{vk}(l,j)\lambda_{vk}(l) T_l}, \quad l = 0, ..., M \tag{5}$$

where $l = p(j)$ or $l = j$ is a leaf node. $c_{vk}(l, j)$ is the access probability of data item k to move from node l to node j, satisfying $\sum_{j \in \boldsymbol{b}(l)} c_{vk}(l, j) = 1$ and $0 \le c_{vk}(l, j) \le 1$. When $l = j$ is a leaf node, $c_{vk}(l, j) = 1$.

Based on the above definition of non-uniform access, we utilize an $M + 1$ states discrete-time Markov chain (DTMC) to model the LRU-$a(\boldsymbol{m})$ replacement policy for data item k at time t^n, denoted by $(Y_k)_n, n \ge 0$, where Y_k is the node id keeping data item k. The state space is represented by $\{0, ..., M\}$ and we thus give the steady state of data item k in cache node j as

$$\pi_{k,j} = \frac{\prod_{(l,l') \in P_k(j)} \alpha_k(l, l')}{\Omega(\boldsymbol{m})} \tag{6}$$

where $\Omega(\boldsymbol{m})$ is the normalizing constant ensuring that $\sum_{k,j} \pi_{k,j} = 1$, and $P_k(j)$ denotes all possible paths for data item k to access node j. The result generalizes prior work in [?] to analyze the steady state of caches with non-uniform access paths for data items, thus offering a solution for layered data access under LRU-$a(\boldsymbol{m})$ policy.

Further, to solve the deterministic time T_l that provides an approximation to LRU-$a(\boldsymbol{m})$, we apply a set of fixed point equations for $T_l, l = 1, ..., M$, similar to the iterative expressions in [? ?]. The equations utilize the average time that data item k spends in node j to derive the probability of data item k residing in node $j \ge 1$ at a random instant. These probabilities are summed up to equal the capacity of each cache node. The key performance measure for the model is

417 denoted by the total miss ratio $p_{miss}^{(v)}$ for stream v as:

$$p_{miss}^{(v)} = \sum_{k} \lambda_{vk}(0) \left(1 - \sum_{j=1}^{M} \gamma_{k,j}\right), \tag{7}$$

where $\gamma_{k,j} = \pi_{k,j}\bar{t}_{k,j} \left(\sum_{i=0}^{M} \pi_{k,i}\bar{t}_{k,i}\right)^{-1}$ denotes the probability of item k in node $j \geq 1$ at a random instant and $\bar{t}_{k,j}$ is the average time of item k spent in node j. When $j = 0$, $\bar{t}_{k,j} = \lambda_{vk}^{-1}(j)$. When $j = 1, 2, ..., M$, it is $\bar{t}_{k,j} = (1 - e^{-\lambda_{vk}(j)})(\lambda_{vk}(j))^{-1}$. The corresponding hit ratio is then given by $p_{hit}^{(v)} = \lambda_v - p_{miss}^{(v)}$.

4 Integrated Caching-Queueing Analysis

In this section, we integrate the modeling of layered data access developed in Section 3 into layered queueing model to establish a new formalism for the modeling of in-memory and computational services. To tackle this, we first define two novel elements, *i.e., cache-task* and *item-entry*, to declare caching entities in an LQN model. In particular, we present the extension for single-node caches ($h = 1$) as they are more widespread, but our implementation also supports $h > 1$ through the formulas above. Then we give the analytical solution for the resulting LCQ models to obtain accurate steady-state performance estimates.

4.1 Layered Caching-Queueing Models

To extend LQN models to include caching components, we first define two novel components as follows:

- *cache-task*: Each caching node is defined as a cache-task in the LCQ model. Each cache-task offers the ability to access a collection of items through a cache, such as a Memcached key-value store. Cache-tasks have the basic properties of tasks of LQN models, but also include four additional properties for caching: the total number of items, the cache capacity, the cache partitioning into nodes, and the cache replacement policy.
- *item-entry*: The services provided by a caching node are defined as item-entries in the LCQ model. Each item-entry represents a collection of items accessible through the cache-task. Item-entries are similar to standard entries, but add the property of the popularity of the items, *e.g.,* Zipf or custom distribution.

Upon issuing a call to an item-entry exposed by a cache-task, a client obtains the requested object after either a cache hit or a cache miss. The hit/miss selection depends on the cached contents and reflects as a choice of a different branch in the activity graph (*i.e.,* DAG) bound to the item-entry. This is modeled through a novel precedence relationship, *i.e.,* an activity graph edge. We name this precedence relationship as *cache-access* for the cache hit and cache miss activities under each item-entry, which can be used to specify the branching in the activity graph. The *cache-access* transforms an incoming job into a hit or miss job, sending out the transformed job to the corresponding outgoing link with a certain probability.

Based on such extensions, we generalize the LQN model by enabling caching nodes and denote the model as an LCQ model. Figure 4 presents an example of an LCQ model containing one cache-task to model the media workflow in Figure 1. The top-right symbol in the task parallelogram distinguishes a cache-task from an ordinary task. When a request issued by the media front-end arrives, the activity A_0 will issue calls to Memcached to retrieve cached contents. We model the probability that the item is in the Memcached with p_{hit} and out of Memcached with $p_{miss} = 1 - p_{hit}$. If a cache hit occurs, the cache hit activity A_{hit} is selected with p_{hit} to issue calls to Media services for further processing.

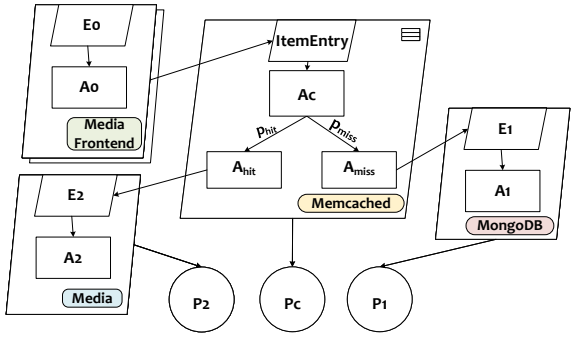


Fig. 4. An example of an LCQ model containing one cache-task

If a cache miss occurs, the cache miss activity A_{miss} is then selected with p_{miss} to issue calls to MongoDB services to retrieve the contents. This generated architecture allows joint modeling of caching and queueing into the same layered model. It is important to note that p_{hit} and p_{miss} need *not* to be specified by the end user. They are automatically determined through the solution of the stochastic models underpinning the LCQ model, which also relies on a stochastic equilibrium analysis of the cache replacement policy, as described in the next section.

4.2 Analytical Solution for the LCQ model

To solve the LCQ model, the first step is to decompose the entire model into a group of sub-models. Each sub-model may be seen as a mixed queueing network that contains clients and servers. The term mixed refers to the simultaneous presence of open and closed service classes. The former represents asynchronous requests from clients, while the latter represents synchronous ones (*i.e.,* closed since issued from a finite connection pool). In particular, when a queueing network contains the cache module, the workload is synchronous, which means the issue of the next request to the cache is required to take place after the reply of the last request issued to the cache.

To model these dynamics, we additionally divide the caching sub-model into two sub-models, as shown in Figure 5. In the upper sub-model, the cache module is isolated in an open queueing network with asynchronous Poisson streams. While in the lower sub-model, the delay and the queueing station are contained in a closed queueing network. To solve the caching sub-model, we generalize the analytical method for non-layered queueing networks from [**?**] to the multi-layer setting of LCQs.

Cache requests arrive in the upper sub-model as multiclass Poisson streams with rate $\lambda_v = \sum_r \lambda_{vr}$. Each stream maps to one of the chains in the original caching sub-model. We use an initial guess of the arrival rate $\lambda_v^{(0)}, v = 1, ..., u$ and of the hit and miss rates of the cache node, *i.e.,* $p_{miss}^{(v,t)}$ and $p_{hit}^{(v,t)}$. The obtained values in each iteration are leveraged in [**?**] to update the arrival rate $\lambda^{(t+1)}$ by the fixed point iteration:

$$\lambda_v^{(t+1)} = \frac{s_v \lambda_v^{(t)}}{\lambda_v^{(t)} \theta_t + p_{miss}^{(v,t)} \theta_m + p_{hit}^{(v,t)} \theta_h}, \tag{8}$$

where $\theta_t, \theta_m, \theta_h$ respectively are the mean delay between consecutive requests to the cache, and the mean delay due to miss and item fetch. Here, s_v is the maximum number of pending requests of stream v to the cache, *i.e.,* the total population of jobs in the caching sub-model. The iteration process continues until the value of λ_v converges.

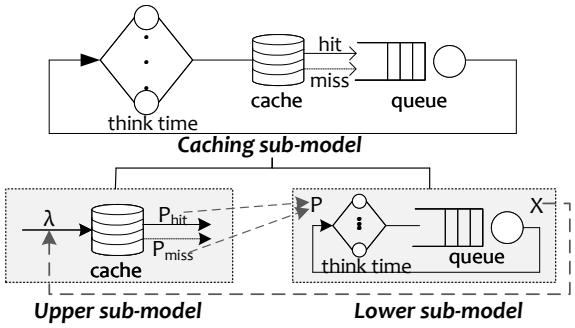


Fig. 5. Decomposition of the caching sub-model in the LCQ model

The hit and miss rates at each iteration are used to parameterize the queueing network of the *lower* sub-model as routing probabilities. That is jobs representing calls that experience cache hits will dynamically switch after visiting the cache to a *hit service class*, which will be processed by the queueing station according to the activities specified in the activity graph for hits. Similarly, calls that experience misses will switch to a *miss service class* representing the workflow in the activity graph for misses. This class-switching based separation of the hit and miss calls enables the modeling of the different response times experienced by the two classes of requests, which depend on whether the items they accessed were cached or not. At the end of the chain of operations that follow hit/miss, both classes of jobs will be merged back into the original class and return to the user as a response for that class.

The solution to the upper sub-model is reconciled with the lower sub-model iteratively, as presented in line 6-15 in Algorithm 1. After an initialization of the arrival rate $\lambda_v^{(0)}$ (*e.g.,* to a lower bound such as $1/(\theta_t + \theta_m + \theta_h)$), miss probability $p_{miss}^{(v,0)}$, and hit probability $p_{hit}^{(v,0)} = 1 - p_{miss}^{(v,0)}$, the iterative approach begins by solving the upper sub-model with (8) to calculate $\lambda_v^{(n)}, n \geq 1$. The hit and miss probabilities, *i.e.,* $p_{hit}^{(v,n)}$ and $p_{miss}^{(v,n)}$, are then solved with approximation methods, *e.g.,* the fixed-point iteration in (3). These probabilities are passed to the lower sub-model to set the routing matrix. Subsequently, the lower sub-model undergoes analysis with a queueing network solver method $f(s)$, *e.g.,* approximate mean value analysis (AMVA) [**?**], to obtain the mean throughput $X_v^{(n)}$. The computed throughput is then passed to the upper sub-model as the next arrival rate $\lambda_v^{(n+1)}$. This iterative process continues until the arrival rate λ_v converges.

The entire procedure to solve the LCQ model containing multiple cache-tasks is presented in Algorithm 1, where $f(\cdot)$ is the custom solver for the sub-model s that returns throughputs, queue-lengths, response times, and utilizations. Specifically, when presented with an input of the LCQ model containing cache-tasks, our approach initially decomposes the entire model into sub-models using loose layering [**?**]. In cases where a layer encompasses a caching module, we further partition the caching sub-model into upper and lower sub-models, updating parameters at each iteration such that the solution of the upper sub-model is reconciled with that of the lower sub-model. The iteration process continues until the value of λ converges. Alternatively, for layers without a caching module, we employ custom solvers, such as AMVA, to address the respective sub-model accordingly in line 17-18, which follows the standard LQN solution algorithm explained in [**?**]. Based on this, we can obtain the mean performance metrics of the model, such as response times and throughputs.

Algorithm 1 Solution algorithm for the LCQ model

Input: LCQ model with cache-tasks, Sub-model solution algorithm $f(\cdot)$

Decompose the LCQ model into S sub-models
while not converged or not reached iteration limit **do**
  for $s \leftarrow 1$ to S **do**
    if s **is** caching sub-model **then**
      Decompose sub-model s into two sub-models.
      Initialize $\lambda_v^{(0)} \leftarrow \frac{1}{\theta_t+\theta_m+\theta_h}$, $p_{miss}^{(v,0)} \leftarrow \frac{1}{h+1}$, and $p_{hit}^{(v,0)} = 1 - p_{miss}^{(v,0)}$ in the upper sub-model.
      $n = 1$
      while $||\lambda_v^{(n)} - \lambda_v^{(n-1)}|| \geq \delta$ **do**
        Solve the upper sub-model for $\lambda_v^{(n)} = \frac{s_v \lambda_v^{(n-1)}}{\lambda_v^{(n-1)}\theta_t + p_{miss}^{(v,n-1)}\theta_m + p_{hit}^{(v,n-1)}\theta_h}$.
        Using $\lambda_v^{(n)}$, obtain $p_{hit}^{(v,n)}$ and $p_{miss}^{(v,n)}$ with (3) or (7).
        Pass $p_{hit}^{(v,n)}$ and $p_{miss}^{(v,n)}$ to the lower sub-model to set its routing probabilities.
        Solve the lower sub-model with f.
        Return the throughput returned by f to the upper sub-model as the next arrival rate $\lambda_v^{(n+1)}$.
        $n = n + 1$
      end while
    else
      Obtain the response time for sub-model s using the solution algorithm f.
      Update in all other layers the think time for requests that originate from sub-model s.
    end if
  end for
end while
Obtain using f response times, residence times, throughputs, and utilizations at each layer.

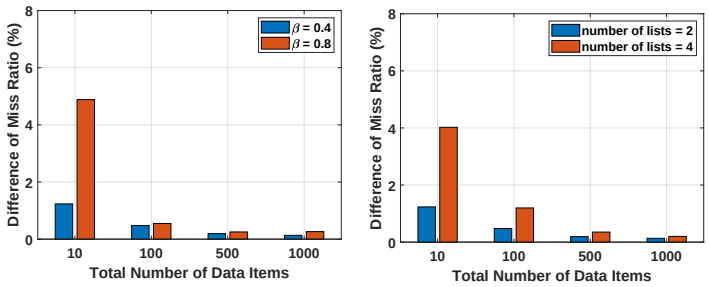
Output: LCQ response times, residence times, throughputs, and utilizations.

5 Model Validation

5.1 Hierarchical Caching Model Validation

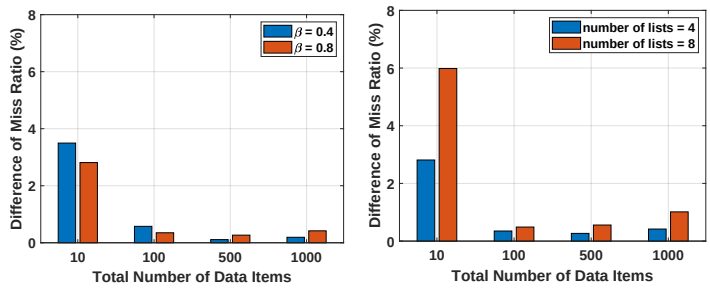
We validate our proposed analytical model for hierarchical cache modeling considering the LRU-$a(\boldsymbol{m})$ replacement policy. For caches employing the RR-$a(\boldsymbol{m})$ replacement policy, we can draw upon the validation process outlined in [**?**], leveraging FPI and singular perturbation approximation (SPA). Since our model generalizes prior research [**?**], which has achieved asymptotically exact results for linear caches, we extend the validation to encompass parallel and tree structures respectively, catering to a wider array of practical scenarios.

For parallel-structured caches, which can be regarded as a special case of tree-structured caches. We consider a total number of 10, 100, 500, and 1000 data items. The ratio of the total cache capacity to total data items β is set as 0.4 and 0.8. The distribution of data items obeys Zipf distribution with a parameter of 0.8. We assume the distribution of data items obeying Zipf distribution, which is widely used for approximating request patterns in several measurement studies, e.g., [**? ?**]. We choose a Zipf parameter of 0.8 as it is within the recommended range (0.64 to 0.85) identified in [**?**] for accurately fitting various request traces and also used in other validation studies, e.g., [**? ?**]. The results of comparing the miss ratio for different cases are illustrated in Figure 6. We can observe that the errors between simulated and analytical results are no larger than 5% for all cases. Moreover, the error declines with larger-scale data items under the same number of lists or β, which is conducive to analyzing practical situations.



(a) Same number of lists (b) Same $\beta = 0.4$

Fig. 6. Comparing the miss ratio of parallel-structured caches between analytical and simulated methods



(a) Same number of lists (b) Same $\beta = 0.8$

Fig. 7. Comparing the miss ratio of tree-structured caches between analytical and simulated methods

Next, for tree-structured caches, *i.e.,* different lists are connected linearly and/or in parallel, we consider a total number of 10, 100, 500, and 1000 data items obeying Zipf distribution with a parameter of 0.8. The ratio β is also set as 0.4 and 0.8. From Figure 7, it can be seen that, for all cases, the errors between simulated and analytical results do not exceed 6%. Under the same number of lists, the error is declined with the increase of the number of data items. Under the same ratio β, the error is increased when there are more lists.

Overall, our analytical model for hierarchical caches shows high accuracy and generality in various caching architectures.

5.2 Validation Under Real Traffic

In this section, we illustrate our analytical model using trace-driven simulation. Specifically, we use the temporally-correlated 2008 YouTube trace from [?], comprising 611, 968 requests. This is representative of real multimedia content that may realistically be cached near the edge, *e.g.* for download by user mobile devices. The chosen trace encompasses 303, 332 distinct items and originates from 16, 337 client IPs, each mapped to a single stream v. The autocorrelation of the unshuffled and shuffled Youtube trace data is shown in Figure 8, showing a high positive correlation for the original (unshuffled) data. The shuffled trace retains the same distribution of inter-arrival times as in the original trace, but

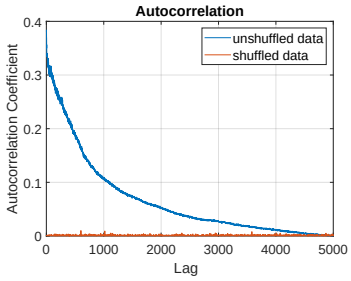


Fig. 8. Autocorrelation of the original and shuffled 2008 Youtube trace

Table 1. Comparing the miss ratio difference between the analytical method and trace-driven simulation. β is the ratio of cache capacity to the number of items.

number of lists	list capacity $m_l, l = 1, 2, ...$	β	Miss Ratio			Miss Ratio Difference	
			analytical	simulated (unshuffled)	simulated (shuffled)	unshuffled data	shuffled data
2	1516	0.01	0.4517	0.4783	0.4535	0.0266	0.0018
4	758		0.4411	0.4725	0.4569	0.0314	0.0157
2	3032	0.02	0.4310	0.4613	0.4336	0.0303	0.0026
4	1516		0.4184	0.4541	0.4402	0.0357	0.0218
2	7583	0.05	0.3907	0.4253	0.3954	0.0346	0.0047
4	3791		0.3756	0.4166	0.4081	0.0410	0.0325
2	15166	0.10	0.3453	0.3832	0.3525	0.0379	0.0072
4	7583		0.3291	0.3741	0.3703	0.0450	0.0412

removes the autocorrelations, thus helping to assess the impact of temporal dependence alone on the applicability of our models, which rely on IRM assumptions.

Our simulations employ a cache with either linear or tree structure under the LRU-$a(\boldsymbol{m})$ replacement policy. Specifically, for tree structures with 4 nodes, direct connections exist between lists 1-2, 1-3, and 2-4. The ratio β of total cache capacity to the number of the readable data items is set at 0.01, 0.02, 0.05, and 0.10. Results are given in Table 1.

For each simulation, we calculate the total miss ratio and compare it with the estimates generated by our analytical model. Across all scenarios, the average *difference* in miss ratio for unshuffled data stands at 0.0353, while for shuffled data, it reduces to 0.0159. These results highlight that the proposed model can display modest differences in miss ratio also under significant violations of the IRM assumption.

5.3 Generalized LCQ Model Validation

We validate the generalized LCQ model by Java Modelling Tools (JMT) [**?**], which is an open source suite for performance evaluation. Specifically, we use the JSIMgraph discrete-event simulator, which facilitates the analysis of queueing networks and stochastic Petri nets. Stochastic Petri nets [**?**] describe a set of tokens, equated in our model with jobs or data items, circulating across places, represented diagrammatically as circles. Transitions, represented as boxes, are defined by exponential timers that, upon elapsing, pull jobs from places connected to their input and deposit a set of newly-generated tokens to the places connected to their output.

As the RR replacement policy involves a synchronized movement of randomly chosen items among lists, this can be encoded in terms of transitions that pull and deposit tokens from the cache, each representing a data item access. The support in JMT for queueing networks allows us instead to characterize the queueing behaviour and CPU consumption

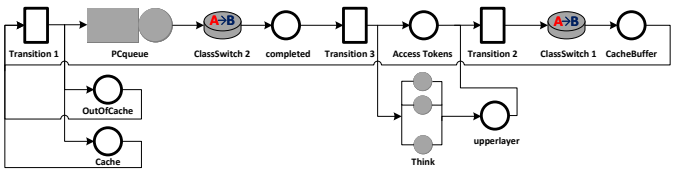


Fig. 9. Validation model combining queueing network and stochastic Petri net nodes

associated to cache service. The joint assessment of caching and queueing then follows by simulating a hybrid model combining both queueing stations and stochastic Petri nets. From this, we can get an accurate estimate of the expected performance of our proposed LCQ model, against which we compare our FPI-based implementation.

In particular, we consider a validation model containing the upper two layers of the model shown in Figure 4. In the upper layer, the underlying processor P_1 adopts the processor sharing strategy and the number of users is represented by the multiplicity of the reference task RT_1. The think time is set as an Immediate transition, which fires instantaneously once its enabling condition is satisfied, to ensure that the client issues without delay the request to the cache-task.

In the lower layer, the capacity of the cache-task is set as 1 with a total number of 3 items. All the items obey discrete uniform distribution and take the RR strategy. The underlying processor P_c also employs the processor sharing strategy and the number of tokens in this layer is denoted by the multiplicity of the cache-task. The service rate of hit and miss activities on processor P_c is exponentially distributed with a parameter of μ_h and μ_m respectively.

The validation model is mapped to the JMT model presented in Figure 9. The jobs first originate from the delay node and are stored at the place of *upperlayer*. Then, utilizing Petri net transition, *i.e.*, waiting for the tokens to be available in the inbound places and firing a given number of tokens in the output, we enable the transition 2 by 1 job from the upper layer and 1 token, to fire 1 job into the lower layer. To determine a cache hit or miss in the lower layer, we regard the cached items as different job classes. Thus, the fired job class can switch into any of the items at the *ClassSwitch 1* node according to the probability distribution. Based on the state of the stored items in *Cache* and *OutOfCache* places, the transition 1 determines whether to fire a hit or miss job class into the queue. After different services for the hit or miss job at the queue node, the job transforms back to the original job class at the *ClassSwitch 2* node and enables the transition 3. Finally, the transition 3 fires 1 job to the place of *Access Tokens* and 1 job to the delay node to execute iteratively.

We simulate the above model with JMT to validate the analytical model. When considering an equal number of users and tokens, we assume the total users as 2, 5, 10, 20, 50, and 100. When considering the number of tokens is half that of the users, we assume the total users as 2, 4, 8, 10, 40, and 100. The service rate μ_h and μ_m are set as 1 and 0.5 respectively. The performance metric we select is the residence time at the queue. This corresponds to the total time between successive visits to the think time station, since the cache replacement transitions are instantaneous.

The results indicate the differences of residence time between the analytical and simulated model under different combinations of numbers of users and tokens, as shown in Figure 10. It shows that the differences are not exceeding 8.26% of residence time in all cases, which proves the high accuracy of the generalized LCQ model. The simulation model adheres to rules governing cache replacement algorithms, ensuring accurate emulation of cache behavior. Our

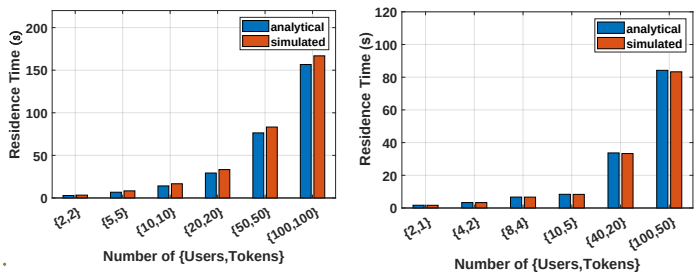


Fig. 10. Comparison of residence time between analytical and simulated methods

approach, while not directly employing CTMC due to the potential for state space explosion, draws upon its principles to closely mirror these dynamics with a notable degree of accuracy.

6 Application: Edge Service Placement

6.1 Problem Statement

Nowadays the combination of microservice architecture with edge computing has attracted much attention [**?**]. In practice, the processing of many microservice applications requires both computation and data at edge nodes. For example, in video analytics [**?**], the request is first forwarded to the content caching function to retrieve the cached contents. After the identification, the request is routed to video compression and video analytics services for further processing.

To evaluate the performance of applications of this kind, we illustrate the benefits of our methodology by considering a cluster of M edge nodes connecting N users in the edge computing system. Each edge node provides multiple services in co-located edge servers and is equipped with a shared cache node that stores data for these services.

Each user offloads several jobs to the connected edge node to be served. We assume the maximum number of jobs of running jobs to be K, and the total number of services to be C. When a job is processed, it first checks if the required data is available in the shared cache node. The shared cache node follows the RR replacement policy and has a capacity of q. If so, the data is retrieved from the cache, and then the job is served by the corresponding services provided by the edge node. If not, the data has to be retrieved from other cache nodes or remote data centers with a longer processing time before being served.

The offloaded jobs may have precedences that must be processed in order. For example, a face recognition application [**?**] can be regarded as consisting of five successive jobs, *i.e.,* object acquisition, face detection, pre-processing, feature extraction, and classification. Each job can be operated solely at the edge node that provides corresponding services. When the requested services are not provided in the connected edge node, the jobs need to be forwarded to other edge nodes that satisfy the requirements. Furthermore, different types of jobs at the same node also obey sequential processing.

For job k at node m, $k = 1, 2, ..., K$, $m = 1, 2, ..., M$, we define two parameters as follows:

- *Service Placement Decision* p_{mk}. If node m provisions the service requested by job k, $p_{mk} = 1$. Otherwise, $p_{mk} = 0$. The set of all nodes that provision the service for job k is denoted as $\boldsymbol{F}_k = \{m | p_{mk} = 1, m = 1, 2, ..., M\}$.

- *Mean service time* t_{mk}. This is the amount of computational processing requested by job k upon visiting node m. Different jobs may request different mean service times.

In the design and deployment of such edge service placement applications, minimizing computational overhead while satisfying the constrained resources is paramount. To address this, we introduce our optimization strategies based on the proposed LCQ model, aiming at predicting the minimized response time cost incurred by the application. This serves as a baseline for efficient evaluation of design-time applications, addressing data caching and processing simultaneously. The optimization process is conducted as part of the initial design phase or during periodic design updates, depending on factors such as changes in application requirements.

Specifically, our optimization objective is to minimize the end-to-end total response time R of the LCQ-based model, which is expressed as

$$R = \sum_{i=1}^{C} \frac{X_i}{X} R_i, \tag{9}$$

where X_i, R_i, and X are the throughput of job class i, the response time of job class i, and the total throughput summed over all service classes, respectively. The ratio X_i/X may be seen as related to the probability at steady-state of an arrival being of class i. Thus, the problem is formulated as

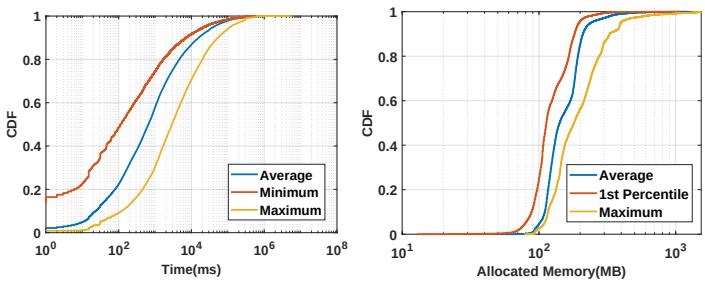
$$\begin{aligned} \min_{\boldsymbol{x}} \quad & R(\boldsymbol{x}), \\ s.t. \quad & \sum_{m \in F_k} x_{mk} = 1, \\ & x_{mk} \in \{0, 1\}, \end{aligned} \tag{10}$$

where x_{mk} denotes the scheduling decision for job k at node m. The vector $\boldsymbol{x} = [x_{11}, ..., x_{mk}, ..., x_{MK}]$ represents the set of scheduling decisions for all jobs and $R(\boldsymbol{x})$ is the corresponding system response time with decision vector $\boldsymbol{x}$. The first constraint guarantees that job k is solely served at one (and only one) feasible edge node provisioning the requested services. The second constraint ensures that if job k is scheduled to be served at node m, then $x_{mk} = 1$, or $x_{mk} = 0$ otherwise. The program complexity is dominated by the number of variables, which grows as $O(M \cdot K)$. We solve the optimization problem by a genetic algorithm (GA) solver. The detailed configuration of the GA is given in Appendix C.1.

6.2 Experimental Evaluations

To evaluate the generalized LCQ model to real workloads, we perform a trace-driven simulation to demonstrate the system applicability in principle to production systems. The trace is collected from a subset of applications running on Azure Functions in 2019 [**?**]. We focus on a random subset of the trace, which contains 34385 application ids and the corresponding serverless function ids belonging to the same application. The invocation rate and the distribution of execution time for each function as well as the distribution of memory usage for each application are also given in the trace.

Figure 11(a) shows the cumulative distribution of the minimum, average, and maximum execution time for all functions. We can see that 50% of the functions have average and maximum execution time less than 0.7s and 3s. 90% of the functions execute at most 50s and 95% of the functions take less than 50s on average. Figure 11(b) presents the cumulative distribution of the 1st percentile, average, and maximum virtual memory reserved for all applications. Each



(a) CDF of Execution Time (b) CDF of Allocated Memory

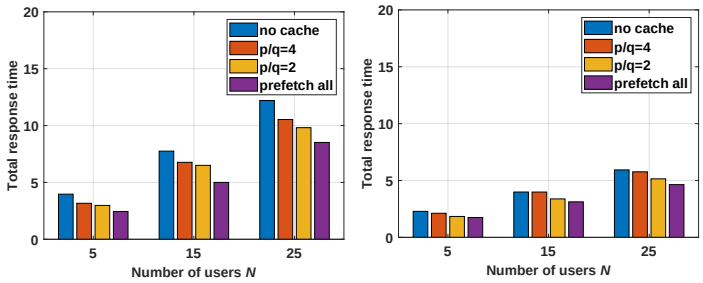
Fig. 11. Properties of the selected subset of the real-world Azure traces [?]

application calls for one or more functions. We can observe that 99% of the applications consume no more than 400MB on average and 99% of the applications are allocated at most 1000MB.

To simulate the trace-driven data, we implement the generalized LCQ model in LINE [?]. To set up the parameters for simulations, we configure the cache capacity m for each node as 250, 750, 1024 MB, according to a standard Redis caching package offered by Microsoft Azure [?]. Corresponding cached items obey a Zipf distribution with parameter $\eta = 1.0, 1.4$ and are assumed of identical size, the total size n of which is proportional to the node cache capacity as 0.5, 1, 2, 4, 8 GB. The number of edge nodes M and number of services C scales up to 16 and 40 respectively. The distribution of service time is assumed as exponential with a mean value same to the average execution time collected from the Azure traces. The number of users is configured as 5, 15, 25 considering the capacity of each edge node [?]. For each user, the probability to request a certain service is determined by the invocation rate of the service.

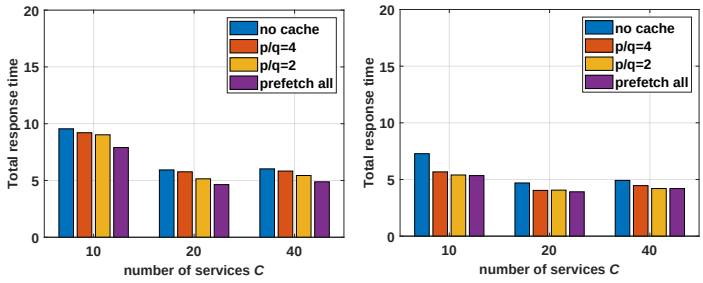
For each combination of the parameters, we generate 30 models with different random seeds. In each model, different services on each node are added with cache-tasks that specify the size of total items and the allocated cache capacity for each service. The sum of service capacities is equal to the node cache capacity, which guarantees the maximum utilization of the constrained in-memory storage for each edge node. We employ the total system response time as the performance measure and compare our proposed scheme to the case of *no cache* and *prefetch all*. The *no cache* scheme deploys no caching module that all the jobs need to wait for the requested contents from origin servers before being processed locally. While the *prefetch all* scheme indicates each edge node caches all the items with a full utilization of the local memory.

From Figure 12 (a) and (b), we can see that the total response time increases with the growth of the number of users under same conditions, which attributes to the cumulative queueing time. With same number of users, *no cache* scheme as the baseline exhibits the highest response time because jobs need to obtain required contents from the origin server. Conversely, *prefetch all* scheme exhibits the lowest response time but ignores the large memory consumption. Compared to the *prefetch all* scheme, although our proposed scheme shows an increase of the response time at most 35% and 30%, but reduces the memory usage by 500MB and 250MB respectively. This is because our proposed system optimizes the allocation of the cache capacity for each service based on the job scheduling and service placement strategies on different edge nodes, which achieves to maximize the utilization of limited edge resources. Overall, the total response time declines significantly when the number of nodes rises, which is benefit from more concurrent servers that bring down the response time.



(a) $M = 2, C/M = 5, \eta = 1$ (b) $M = 4, C/M = 5, \eta = 1$

Fig. 12. Total response time with respect to the increase of the number of users N (Azure dataset)



(a) $M = 4, N = 25, \eta = 1.4$ (b) $M = 8, N = 25, \eta = 1.4$

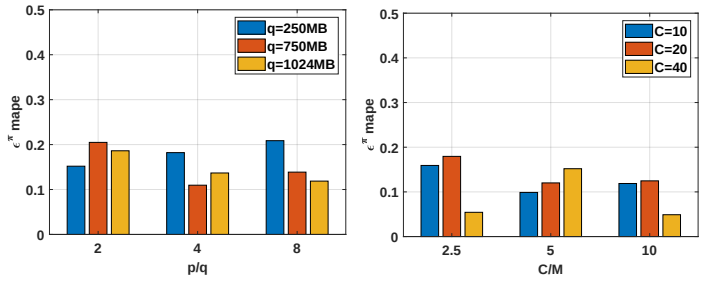
Fig. 13. Total response time with respect to the increase of the number of services C (Azure dataset)

From Figure 13 (a) and (b), we can see that the total response time decreases with the growth of the number of services, but tends to be stable after C reaches 20. The decreasing trend in the first stage is the case when the number of services is less than the number of users. Under the circumstances, the possibility of users requesting the same service is higher, which results in a longer queueing time for the same server. As the number of services grows to equal to or larger than the number of users, the possibility of calling the same service reduces, thus leading to a flat trend in the later stages. With same number of services, our proposed scheme only sees an increase of the response time at most 24% and 14%, but with a less memory usage of 500MB and 250MB compared to the *prefetch all* scheme.

Further, we analyze the sensitivity to p/q and C/M ratio on these experiments. We focus on the miss ratio and define the performance metric as

$$\epsilon^{\pi}_{mape} = \frac{1}{Q}\frac{1}{M}\frac{1}{C}\sum_{k=1}^{Q}\sum_{j=1}^{M}\sum_{i=1}^{C}\left|1 - \frac{\hat{\pi}_{i,j}}{\pi_{i,j}}\right|, \tag{11}$$

where $\hat{\pi}_{i,j}$ is the estimated miss ratio for service i at node j. The services that are not requested by users on each node are not included. From Figure 14 (a), we can see that the average percentage error ϵ^{π}_{mape} for p/q ratio approximately ranges between 0.1 and 0.2. When the node cache capacity is high, the error metric decreases with the size of total items increases. From Figure 14 (b), we can observe that the average percentage error ϵ^{π}_{mape} for C/M ratio approximately



(a) $M = 8, N = 25, C = 40$ (b) $N = 25, p = 0.5, q = 250$

Fig. 14. Sensitivity to p/q and C/M ratio with $\eta = 1$ (Azure dataset)

ranges between 0.05 and 0.15. The results demonstrate the robustness of our proposed scheme in effectively managing substantial data volumes and diverse levels of edge computing processing capabilities.

Concerning time and space requirements for the execution of the proposed scheme, the above experiments are conducted on an AMD EPYC 7302P 16-Core Processor. For each replication, the estimate of memory usage and execution time are no more than 4.45MB and 0.2s, which reflects our proposed scheme is scalable for multiple workloads.

7 Application: Data Mesh Design

7.1 Data Mesh Architecture

Data mesh, a novel paradigm of data architecture, has recently gained attention [**?**]. It decentralizes data management by shifting the responsibility from a centralized data team to distributed domain teams. Each domain team takes responsibility for their own data, such as processing source data or publishing data products. This enables each domain team to leverage its specialized knowledge and expertise to deliver more insightful and valuable views.

Microservices architecture aligns well with this distributed approach, as each microservice can correspond to a specific domain team and encapsulate the logic and functionality related to that data domain. It allows each domain team to independently develop and deploy their data services, which can be designed to handle specific data processing, transformation, and integration tasks, thereby enabling the team to have control over their data operations.

7.2 Problem Statement

For data products frequently queried by users, samples are copied from data buckets and stored in caches (*e.g.*, Redis or Memcached). Due to geography-based compliance issues, security issues, and organizational constraints, data products of different domain teams may only have access permission to a subset of cache nodes. Different architectural designs of the storage services can be considered on top of this model:

- single-layer architecture: a centralized layer for caches with multiple data buckets located in different regions.
- hierarchical architecture: multi-level layers for caches, that may be distributed at different locations (*e.g.*, network edges) based on user requirements, with multiple data buckets located in different regions.

Considering the different requirements of each domain team, how to properly access and process data products is a major challenge. To tackle this, we present optimization strategies tailored for data product management within the framework of our proposed hierarchical data distribution model. Our goal is to minimize operational costs during the design phase of data mesh applications. Such activities are normally conducted prior to implementation, when the optimal architecture needs to be identified by system engineering teams. A key benefit of our approach compared to simulation is to enable a faster evaluation of individual scenarios by means of the proposed LCQ models.

7.2.1 Hierarchical Data Caching For the hierarchical data caching (HC), our goal is to minimize the total operational cost $C_1(\boldsymbol{x})$, which is set as the total miss ratio of all data products. We define $\boldsymbol{x} = [x_{11}, ..., x_{ij}, ..., x_{IM}]$ as the *caching plan*, where x_{ij} is the access probability of data product $i = 1, \ldots, I$, to be cached at cache node $j = 1, \ldots, M$, satisfying $0 \leq x_{ij} \leq 1$.

Using the previous definitions, we can define the total operational cost $C_1(\boldsymbol{x})$, *i.e.,* the total miss ratio under caching plan $\boldsymbol{x}$, as:

$$C_1(\boldsymbol{x}) = 1 - \boldsymbol{Q}\boldsymbol{\gamma}(\boldsymbol{x}), \tag{12}$$

where matrix $\boldsymbol{Q} = \{q_{ij}, i = 1, ..., I, j = 1, ..., M\}$ and vector $\boldsymbol{\gamma}(\boldsymbol{x}) = \{\gamma_{ij}, i = 1, ..., I, j = 1, ..., M\}$ describe the request rates and the probability of each data product at each cache node under the caching plan $\boldsymbol{x}$, respectively. Thus, the problem of configuring the data caching layers is formalized as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & C_1(\boldsymbol{x}), \\ s.t. \quad & \sum_i s_i \gamma_{ij} \leq m_j, \qquad \forall j \\ & \sum_{j \in \boldsymbol{a_i}} x_{ij} = 1, \qquad 0 \leq x_{ij} \leq 1, \forall i \end{aligned} \tag{13}$$

where the first constraint guarantees that the total size of cached data products at each cache node is no larger than the node capacity. The second constraint limits the sum of the probabilities of data product i to access all child nodes that meet the geographical constraints equal to 1. The complexity of the program depends on the optimization algorithm chosen for its solution and it is driven by the number of variables, which is $O(I \cdot M)$, and the $O(M)$ constraints. Upon using a metaheuristic, such as GA, implemented by embedding the constraint evaluation in the objective calls, this overall results in an estimated complexity of $O(P \cdot G \cdot I \cdot M)$, where P is the GA population and G the number of GA generations. In our experiments, the population P is proportional to the number of variables, making the solution approach overall quadratic in $I \cdot M$. In cases where this exceeds the desired complexity, keeping a fixed GA population as the problem size grows can help to reduce the computational burden.

7.2.2 Hierarchical Caching and Processing For hierarchical data caching and processing networks (HCP), our optimization goal is to minimize the mean SLA deviation of all data products $C_2(\boldsymbol{x})$, *i.e.,*

$$C_2(\boldsymbol{x}) = \frac{1}{I} \sum_i max\{0, R_i(\boldsymbol{x}) - SLA_i\}, \tag{14}$$

where the SLA deviation of each data product is the difference between the actual response time $R_i(\boldsymbol{x})$ occurring for data product i by configuring the system with plan $\boldsymbol{x}$ and the SLA target SLA_i on mean latency allowed for serving data product i. The constraints are same to the program (13).

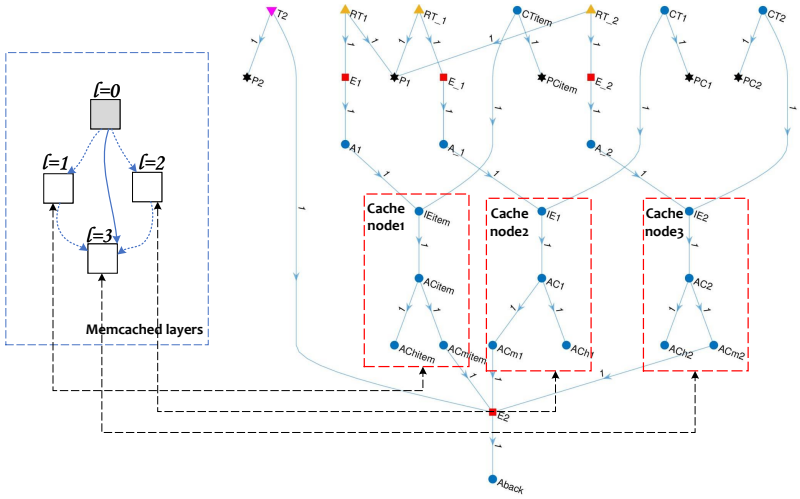


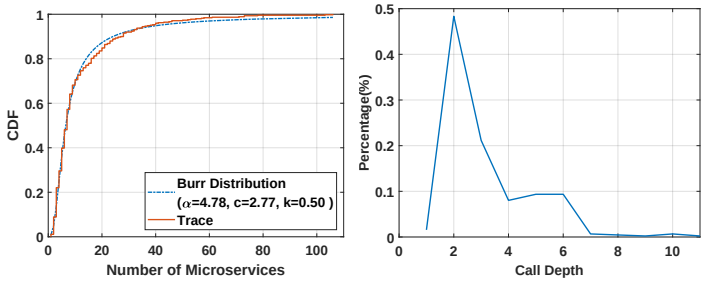
Fig. 15. An example of the transformation from the data caching model to the integrated data caching and processing model *(The yellow triangles, pink triangles, red squares, blue circles and black hexagrams represent reference tasks, tasks, entries, activities and processors respectively.)*

Determining a caching plan for the HCP program is challenging due to the scale of data products involved, which could be in the order of several hundred or thousand products. To overtake this issue, we propose to determine the caching plan using the simplified hierarchical caching model and then integrate the results into a model that includes queueing abstractions to forecast the resulting performance. The detailed implementations are as follows.

By optimizing the program in (13) for the hierarchical caching model, we obtain a caching plan $\boldsymbol{x}$ mapping each data product to access a cache node with a certain probability so as to minimize the total operational cost. The obtained caching plan $\boldsymbol{x}$ is then transformed into a series of LCQ models that describe data caching and processing for each data product. An example of the transformation from the data caching model to the integrated data caching and processing model is presented in Figure 15. The left figure shows users requesting data products from different cache nodes without queueing ($l = 1, 2, 3$). Users requesting data products and each cache node are then transformed into the reference task ($RT1$) and cache tasks (Cache nodes 1, 2, 3) within the LCQ model showcased in the right figure.

Call LCQ(i) the model, obtained after the above transformation, that describes the performance of data product i. In such a model, users that request data product i are denoted in the LCQ by a dedicated reference task, *i.e.,* $RT1$. Such task models the process of requesting data product from the data mesh system through activity, *i.e.,* $A1$. The activity in the LCQ model issues a synchronization call to the cache node that stores the data product i, based on the caching plan $\boldsymbol{x}$. The remaining data products $i' \neq i$ are also modeled in LCQ(i), however, this is done in an aggregate fashion by mapping them to the workflow of a client that describes the arrival rate of requests to the cache node they are stored into.

Within such clients, a request to data product i' will be issued in accordance with the total probability of all data products in this group and the relative frequency of requests of i' with respect to this group. Through these mappings, LCQ(i) provides sufficient details to collect response time predictions for accesses to data product i, while only providing aggregate performance measures for the remaining data products. This is generally preferable than



(a) CDF of the number of microservices (b) Call depth distribution of sampled traces

Fig. 16. Statistical characteristics of the sampled Alibaba traces

modeling all individual data products in the same model, which would result in a large number of job classes in the underpinning queueing models, facing scalability issues. Instead, based on the collection of models LCQ(i), we can calculate the mean response time of processing each data product i with lower computational cost.

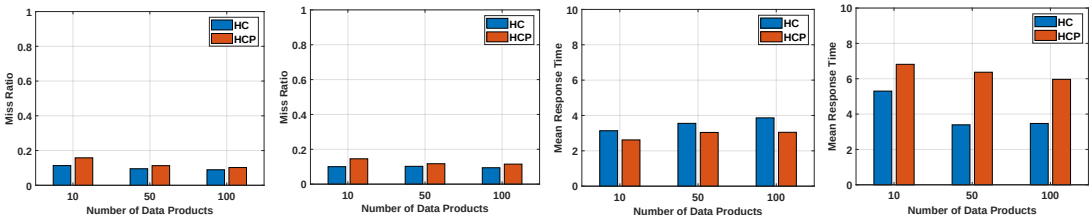
Continuing the description of the transformation, cache nodes are described in the LCQs as cache tasks, such as $CT1$ and $CT2$, with the same settings in the original model used for the caching plan generation, *e.g.,* cache capacity, data product popularity. Attached to each cache task is a workflow that specifies how the cache handles the hits and misses for the requested products. If the requested data product is cached, a cache hit is conducted at the cache node to process the request, *e.g.,* $ACh1$. If the requested data product is not cached, a cache miss occurs and the node forwards the request to data buckets for further processing, *e.g.,* via the call from $ACm1$ to $Aback$.

7.3 Experimental Evaluations

We perform a trace-driven simulation to evaluate our proposed analytical solutions in the data mesh scenario. The trace is collected from a subset of microservices running on Alibaba production clusters over ten thousand bare-metal nodes during twelve hours in 2021 [**?**]. The subset contains 449 trace ids, each of which represents a call graph consisting of multiple calls between different microservices. The total number of involved microservices is 737, which can be categorized into two types: stateful services, *e.g.,* databases and Memcached, stateless services, *i.e.,* data processing for different purposes. The call rate and CPU utilization of each container serving for the corresponding microservice are given.

Figure 16(a) shows the cumulative distribution of the number of microservices for all sampled traces. We can see that the CDF follows a Burr distribution [**?**] with parameters of $\alpha = 4.78, c = 2.77, k = 0.50$. 50% of the traces have less than 10 microservices and 90% of the traces contain no more than 30 microservices. Figure 16(b) presents the distribution of the call depth, *i,e.,* the length of the longest path in a call graph, of all sampled traces. We can observe that 80% of the traces have a call depth of 4 at most and only 2% of the traces possess a call depth over 7.

The parameter settings for simulating the sampled Alibaba traces are configured based on the statistical characteristics. The number of microservices in each call graph is ranging between 2 and 20, while the number of data products is set to 10, 50, 100, and 500. The cached data products follow a Zipf distribution with a parameter value of 0.8, and their total size is proportional to the allocated cache capacity. Each cache node adopts the LRU replacement policy.



(a) Average call graph size = 5 (b) Average call graph size = 15 (c) Average call graph size = 5 (d) Average call graph size = 15

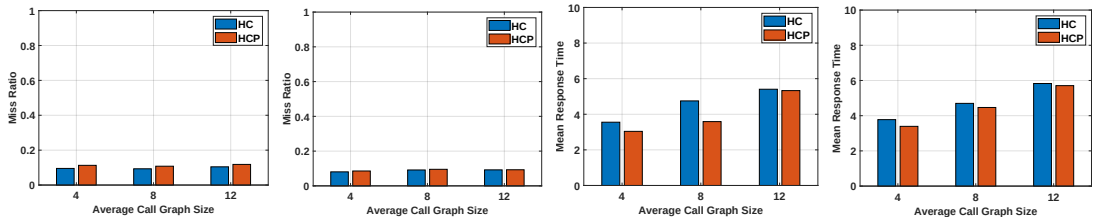
Fig. 17. Miss ratio and mean response time with respect to the increase of the number of data products (Alibaba dataset)

For each combination of the parameters, we first optimize the HC program to obtain a caching plan $\boldsymbol{x}$ with minimized total miss ratio. According to the caching plan $\boldsymbol{x}$, we then transform the plan to LCQ models to calculate the mean response time of processing all data products. Further, with the same hierarchical caching architecture, we optimize the HCP program to derive a caching plan $\boldsymbol{x}'$ with minimized mean SLA derivation. We then utilize the new caching plan to calculate the total miss ratio. Both optimization programs use genetic algorithms (GAs), with similar configurations outlined in Appendix C.1. Thus, we employ the total miss ratio and mean response time as the performance measure and compare the two optimization programs under different cases.

From Figure 17, it is evident that our analytical solution-based optimization of the HC program exhibits a better mean response time for larger scale call graphs while the HCP program performs better with smaller scale. Specifically, the HC program achieves a reduction in miss ratio of 21% on average, accompanied by a significantly lower mean response time of 37% when the average size of call graphs is 15. Conversely, for the average graph size of 5, the HCP program demonstrates an average improvement of 17% in mean response time, albeit at the expense of minor disadvantages in total miss ratio. Also, in such cases, approximately 16% of the services in the HC program fails to meet the SLA requirements. These results highlight the potential benefits of employing analytical solutions to optimize HCP programs for smaller scale call graphs, ensuring compliance with SLA demands. While for systems with higher complexity, leveraging HC programs offers a compelling approach to maintain exceptional performance levels.

From Figure 18, we can observe that, the HCP program outperforms better than the HC program across different scales of data products. Specifically, when considering $I = 50$ and $I = 500$, as the average graph size increases, the HCP program can yield a reduction of up to 24% and 10% in mean response time, respectively. Moreover, it is worth noting that as the average graph size scales up to 12, although the HCP program demonstrates a modest 2% improvement in mean response time compared to the HC program, 26% of the services in the HC program violates the SLA requirements. These findings demonstrate the efficacy of leveraging analytical models to optimize HCP programs across a range of data product variations.

In summary, employing our proposed analytical solutions to optimize HCP program demonstrates better performance in smaller scale call graphs across various data product scales, exhibiting a mean response time improvement of up to 24%. On the other hand, the optimization of HC program showcases higher effectiveness for larger scale call graphs, achieving a significant reduction in miss ratio of 21% and an enhancement in mean response time of 37%.



(a) $I = 50$ (b) $I = 500$ (c) $I = 50$ (d) $I = 500$

Fig. 18. Miss ratio and mean response time with respect to the increase of the average call graph size (Alibaba dataset)

8 Conclusion and Future Work

In this paper, we first introduce the concept of access paths to model hierarchical data caching with constraints on data locality. Then we establish a new class of performance models, termed layered caching queueing (LCQ) models, that generalize layered queueing networks with caching nodes, enabling the joint modeling of in-memory storage and computation services. We provide an exact solution and a time-to-live approximation for caching modeling with RR and LRU policy respectively, along with an analytical solution for numerically analyzing the proposed class of LCQ models.

Our proposed models have been validated and implemented in the LINE tool, and real-world Azure and Alibaba traces are simulated to assess the applicability and scalability of our proposed models in two critical scenarios: edge service placement and data mesh. The results demonstrate that our proposed program achieves a significant improvement of 35% in response time and a memory usage reduction of 500MB compared to baseline heuristics methods for edge service placement. Additionally, our analytical solutions allow for programs to reach up to a 21% reduction in miss ratio and a 37% decrease in mean response time in the data mesh scenario.

In our future work, we propose leveraging the potential of serverless computing as a robust implementation strategy for this research. By harnessing the inherent scalability, elasticity, and cost-efficiency of serverless computing for storage and computation, we aim to optimize system performance and resource utilization, thereby enhancing the overall efficacy of our proposed solution.

A Proof of Theorem 1

The argument relies on showing the reversibility of the underlying Markov process, which can be done following very similar steps as the proof of Theorem 1 in [**?**]. The latter proves the product-form solution of RR-$c(\boldsymbol{m})$, for which the product-form result assumes caches where the parent relationships between nodes (*i.e.*, cache lists) are statically defined and stream-independent. This is also ensured by the assumptions of Theorem 1 as access paths to items are stream-independent. Moreover, while the proof given in [**?**] is for a model that assumes that items enter at cache list 1, the proof does not rely on this assumption, hence the same argument proves the reversibility of the Markov process underlying the RR-$a(\boldsymbol{m})$ policy.

Table 2. Transient and recurrent states under item-dependence costs for the cache in Figure 3.

state s	type	probability $\pi(s)$
$[4,2,1,3]$	transient	0
$[4,1,2,3]$	transient	0
$[3,2,1,4]$	recurrent	$\alpha_{21}\alpha_{12}\alpha_{43}/E(\boldsymbol{m})$
$[3,1,2,4]$	recurrent	$\alpha_{11}\alpha_{22}\alpha_{43}/E(\boldsymbol{m})$
$[2,3,1,4]$	recurrent	$\alpha_{31}\alpha_{12}\alpha_{43}/E(\boldsymbol{m})$
$[2,1,3,4]$	recurrent	$\alpha_{11}\alpha_{32}\alpha_{43}/E(\boldsymbol{m})$
$[1,3,2,4]$	recurrent	$\alpha_{31}\alpha_{22}\alpha_{43}/E(\boldsymbol{m})$
$[1,2,3,4]$	recurrent	$\alpha_{21}\alpha_{32}\alpha_{43}/E(\boldsymbol{m})$

B Example: a cache with access paths

We consider again the model show on the left of Figure 3, assuming $n = 4$ items, $h = 3$ nodes, and $\boldsymbol{m} = (1,1,1)$. We consider $u = 2$ streams that both access items at rate $\lambda_{vi}(j)$. If $\mathcal{P}_{vi}(j) = \emptyset$, item i cannot reach node j under requests from stream v and we indicate this by setting $\lambda_{vi}(j) = 0$. With these definitions we get

$$\lambda_1 = [\lambda_{1i}(j)] = \begin{bmatrix} 4 & 9 & 6 & 5 \\ 10 & 10 & 6 & 0 \\ 8 & 4 & 9 & 0 \\ 0 & 0 & 3 & 8 \end{bmatrix} \qquad \lambda_2 = [\lambda_{2i}(j)] = \begin{bmatrix} 8 & 6 & 2 & 10 \\ 3 & 2 & 8 & 0 \\ 2 & 4 & 5 & 0 \\ 0 & 0 & 2 & 7 \end{bmatrix}$$

Let $C_{vi} = [c_{vi}(l,j)]$ be the access cost matrix for stream v and item i. We assign the following values:

$$C_{11} = \begin{bmatrix} \frac{1}{2} & \frac{1}{6} & \frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad C_{12} = \begin{bmatrix} 0 & 1 & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad C_{13} = \begin{bmatrix} 0 & \frac{7}{10} & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad C_{14} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C_{21} = \begin{bmatrix} \frac{2}{3} & \frac{1}{9} & \frac{2}{9} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad C_{22} = \begin{bmatrix} 0 & 1 & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad C_{23} = \begin{bmatrix} 0 & \frac{7}{10} & \frac{3}{10} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad C_{24} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The irreducible Markov process for the recurrent states has infinitesimal generator

$$Q = \begin{bmatrix} -8 & 0 & \frac{28}{5} & 0 & 0 & \frac{12}{5} \\ 0 & -8 & 0 & \frac{12}{5} & \frac{28}{5} & 0 \\ 15 & 0 & -\frac{39}{2} & 0 & \frac{9}{2} & 0 \\ 0 & \frac{9}{2} & 0 & -\frac{39}{2} & 0 & 15 \\ 0 & \frac{16}{9} & \frac{32}{9} & 0 & -\frac{16}{3} & 0 \\ \frac{32}{9} & 0 & 0 & \frac{16}{9} & 0 & -\frac{16}{3} \end{bmatrix}$$

For example, $Q(6,1) = \frac{32}{9}$ is the transition rate between state $[1,2,3,4]$ and state $[3,2,1,4]$. We determine numerically from the Markov process the following equilibrium state probabilities $\boldsymbol{\pi} = [0.3635, 0.0545, 0.1357, 0.0291, 0.1718, 0.2254]$, where $\boldsymbol{\pi} = [\pi(\boldsymbol{s})]$. We now verify that this is the same result that one would obtain using the product-form expression. Using the definitions in Theorem 1 we first compute the following cost factors

$$\alpha = [\alpha_{ij}] = \begin{bmatrix} 1 & \frac{16}{9} & \frac{32}{9} & 0 \\ 1 & 15 & \frac{9}{2} & 0 \\ 1 & \frac{28}{5} & \frac{12}{5} & \frac{84}{5} \\ 1 & 0 & 0 & 15 \end{bmatrix}$$

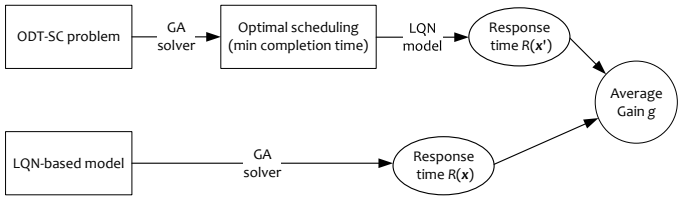


Fig. 19. An overview of the comparative procedure between the LQN-based model and the method in [?]

where rows index items $i = 1, \ldots, n$ and columns index lists $j = 0, \ldots, h$.

With the above parameterization, we obtain from (1) the following state probabilities:

state $\boldsymbol{s}$	*probability* $\pi(\boldsymbol{s})$	*value*
$[3, 2, 1, 4]$	$\alpha_{21}\alpha_{12}\alpha_{43}/E(\boldsymbol{m})$	$\frac{1200}{3301} = 0.3635$
$[3, 1, 2, 4]$	$\alpha_{11}\alpha_{22}\alpha_{43}/E(\boldsymbol{m})$	$\frac{180}{3301} = 0.0545$
$[2, 3, 1, 4]$	$\alpha_{31}\alpha_{12}\alpha_{43}/E(\boldsymbol{m})$	$\frac{448}{3301} = 0.1357$
$[2, 1, 3, 4]$	$\alpha_{11}\alpha_{32}\alpha_{43}/E(\boldsymbol{m})$	$\frac{96}{3301} = 0.0291$
$[1, 3, 2, 4]$	$\alpha_{31}\alpha_{22}\alpha_{43}/E(\boldsymbol{m})$	$\frac{567}{3301} = 0.1718$
$[1, 2, 3, 4]$	$\alpha_{21}\alpha_{32}\alpha_{43}/E(\boldsymbol{m})$	$\frac{810}{3301} = 0.2254$

where we omitted the terms for virtual list 0 since $\alpha_{30} = \alpha_{20} = \alpha_{10} = 1$. The values in the table match the ones obtained numerically for $\boldsymbol{\pi}$.

C Benefits of LQN-based Modeling

To justify the benefits of stochastic LQN-based modeling of distributed workflows, we regard the method in [?] as a baseline for comparison with regards to the response time metric as follows.

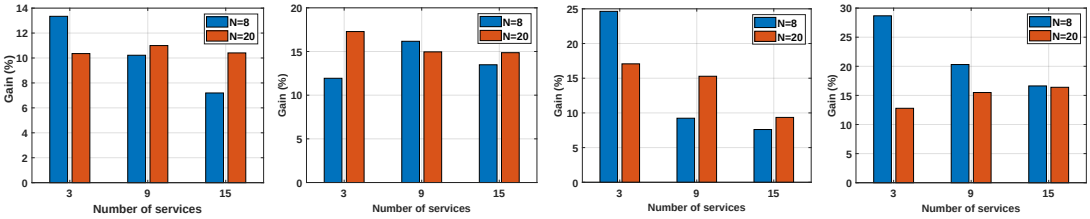
C.1 Experimental Setup

We compare our proposed scheme with the method in [?], which formulates the offloading dependent tasks with the service caching (ODT-SC) problem. The ODT-SC problem considers deterministic scheduling with deterministic service time, while our scheme focuses on stochastic scheduling with random service time. Moreover, the optimization goal of the ODT-SC problem is to minimize the maximum makespan, *i.e.,* the sum of the start time of a job and its execution time, while our goal is to minimize the total response time.

For comparison, we set the mean service time for each job in our LQN-based model equal to the execution time in [?]. The optimization goal of the ODT-SC problem is reset to minimize the total completion time, which is the sum of the makespan of each job. Moreover, the method in [?] reformulates the ODT-SC problem into a convex programming to get feasible solutions. To be more general, we solve the ODT-SC problem by genetic algorithm (GA) without the need to relax the constraints. An overview of the comparative procedure of the two schemes is presented in Figure 19.

Table 3. Parameter settings for the comparative experiments. *nvars* is the number of variables.

Definition	Parameter	Range of Value
number of edge nodes	M	2, 5
number of users	N	8, 20
number of services	C	3, 9, 15
number of iterations / replications	I / Q	30
number of generations	G	1000
number of population	P	$min(max(10 \cdot nvars, 40), 100)$



(a) Exponential Service, $M = 2$ (b) Hyper-exp. Service, $M = 2$ (c) Exponential Service, $M = 5$ (d) Hyper-exp. Service, $M = 5$

Fig. 20. Average gain of response time of the LQN-based model compared to the method in [**?**].

Specifically, we first leverage the GA to address the ODT-SC issue, obtaining the optimal scheduling strategy $\boldsymbol{x'}$of each job. Then the strategy is transformed into an LQN model to calculate the total response time $R(\boldsymbol{x'})$. The response time corresponding to the minimum completion time in each iteration is selected as the comparative object. Meanwhile, we equally exploit the GA to solve the proposed LQN-based model to search for the minimum total response time. Finally, we compare the two response times to get the average gain g of the proposed method, which is expressed as

$$g = \frac{1}{I}\sum_{i=1}^{I}\frac{\parallel R_i(\boldsymbol{x'}) - R_i(\boldsymbol{x}) \parallel}{R_i(\boldsymbol{x'})}, \tag{15}$$

where I is the total iterations.

In each iteration $i = 1, \ldots, I$, to reduce the estimation error, we run Q replications of the GA solver to search for the optimal solutions. In each replication of the GA solver, the number of generations is denoted by G. The dimension of the design variable for the ODT-SC problem is $2K$, of which the first and second K elements are the start times and scheduling decisions of each job respectively. The service time is deterministic but varies in different kinds of services. While for the LQN-based model, the dimension of the design variable is the K scheduling decisions of each job. The service time of each job class is set to be exponentially distributed or hyper-exponentially distributed, of which the mean value is equal to the one in the ODT-SC problem.

A set of experiments are conducted to analyze the performance under different conditions. The setting of different parameters is shown in Table 3 , where *nvars* is the number of decision variables. The distribution of service time is assumed as exponential or hyper-exponential. For simplicity, the user workflow adopts chain structures for multiple services in the experiments, but it is easily extended to cases with parallel structures by LQNs.

C.2 Evaluation Results

The experimental results are presented in Figure 20, which can be seen that using LQN models yields a better average performance than the method in [**?**]. When $M = 2$, the gain of total response time is a minimum of 7%, an average of 12%, and a maximum of 17%. When $M = 5$, the gain of total response time is a minimum of 7%, an average of 16%, and a maximum of 28%. As a whole, the average gain improves with the increase of the number of edge nodes.

Our results show that it is beneficial to apply LQNs to stochastic scheduling as opposed to heuristically applying deterministic scheduling, *i.e.,* the solution to the ODT-SC problem. Therefore, it is meaningful to use the LQN-based modeling for further performance analysis of edge computing systems.

Received November 2023; revised February 2024; accepted July 2024