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

Agrim Bari*, Yuqi Zhou†, and Gustavo de Veciana*
*The University of Texas at Austin, USA
Email: {agrim.bari, deveciana}@utexas.edu
†Purdue University, USA
Email: zhou1168@purdue.edu

***Abstract*— Collaborative VR and social-metaverse applications present a unique challenge for edge caching due to highly correlated client request patterns. Unlike typical web browsing where requests are independent, VR users might navigate spaces in groups. This generates sequential "leader-follower" requests that defy the traditional Independent Reference Model (IRM) used to design most caches. In this paper, we introduce the Grouped Client Request Model to formally capture these dependencies between client requests. Our analysis shows that standard policies like Least Recently Used (LRU) fail to adapt to these dynamics, often evicting content right before a follower needs it. To solve this, we propose Least Following and Recently Used (LFRU), a policy that dynamically learns the causal relationships driving these request patterns. By prioritizing objects based on inferred dependencies rather than just recency, LFRU significantly improves the cache hit ratio. Evaluations using emulated VR traces show that, in structured following scenarios, LFRU achieves up to 2.9× higher hit ratios than LRU and 1.9× higher than LFU.**

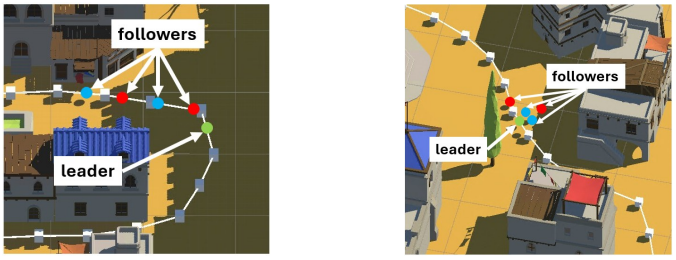
***Index Terms*—Caching, Virtual Reality, Modeling and analysis**

I. Introduction

Managing edge caching for the Social Metaverse. Efficient edge cache management is critical for the Metaverse and VR to ensure low-latency immersion. However, serving multiple users on shared edge resources presents a significant challenge, particularly given the dynamic, multi-user nature of VR workloads. To maximize performance, systems must make two key decisions: what data to place in the cache (placement) and what to evict when capacity is reached (eviction).

Limitations of existing heuristic policies. Traditional policies rely on general heuristics to determine eviction. Least Recently Used (LRU) prioritizes evicting objects accessed least recently, leveraging temporal locality. Least Frequently Used (LFU) retains frequent objects, excelling when content popularity is stable. However, in social VR scenarios, client requests are rarely independent; they exhibit strong inter-user correlations—such as groups moving together—which these standard heuristics fail to capture

Correlated Interactions in Social VR. The Metaverse represents a shift from independent web browsing to immersive, socially integrated experiences. In these 3D virtual networks, clients engage in coordinated activities—such as remote meetings, multiplayer gaming, and collaborative tours—creating highly correlated request patterns that differ fundamentally from the random access patterns assumed by the traditional Independent Reference Model (IRM). For instance, in collaborative edge computing or shared workspaces, actions are often interdependent.



(a) Structured following. (b) Unstructured following.

Fig. 1: Different types of correlations in a VR environment, example relative positions of clients.

In VR environments, these dependencies are driven by social dynamics. As illustrated in Fig. 1, we identify two primary behaviors. *Structured Following* (or Social Guiding, see Fig. 1a) occurs in scenarios with a clear leader, such as a teacher guiding a class through a virtual museum. Here, "followers" replicate the leader's movement path sequentially, creating a deterministic stream of identical content requests. In contrast, *Unstructured Following* (or Group Exploration, see Fig. 1b) arises during looser social interactions, such as friends exploring a virtual city. While clients stay in the general vicinity of a leader, their individual movements are staggered and less predictable.

Bridging Social Behavior and System Design. Current edge caches often fail to support these "Social Metaverse" scenarios because they treat every client request as an isolated event. To bridge this gap, we introduce the *Grouped Client Request Model*. This framework explicitly captures the inter-client correlations driven by social coordination. Furthermore,

we propose a caching policy that dynamically infers these relationships. By recognizing when a "group-following" is occurring, our system prioritizes content for the entire group, ensuring a seamless, low-latency experience for all clients.

A. Related work

Analytical works on caching policies and approximation. We focus on key works in caching policy design and analysis. Early research in caching is summarized in [1], [2], with [3] providing analytical methods for evaluating caching policies. The primary goal of caching policies is efficient cache utilization, typically measured by the cache hit ratio, i.e., the fraction of requests served by the cache. Other objectives include ease of implementation, low operational overhead, and adaptability to fluctuating access patterns. A major distinction among policies is eviction criteria. LRU policy, which updates the cache with the most recently accessed objects, leverages temporal locality. For LRU, under the IRM, the authors in [4]–[8] have obtained an approximation for the hit probability of an object, the fraction of requests for a data object for which the object is in the cache. This approximation has been shown to be accurate as the number of objects scale [6], [8]. We propose a new working-set approximation for hit probabilities under the grouped client request model for LRU.

For stationary request distributions, caching the most frequently requested objects is optimal [9], making Least Frequently Used (LFU) an effective policy. The authors of [10], [11] study LFU and LRU under a semi-Markov modulated request process, where object popularity evolves according to an underlying semi-Markov process, and show that LFU remains optimal even in such settings. However, under dynamically evolving access patterns, LFU may perform suboptimally, highlighting the need for adaptive policies such as LFRU.

Machine Learning-Based Caching Approaches. Machine learning (ML) has influenced the development of adaptive caching strategies. Works such as [12]–[14] use supervised learning to predict content popularity. While effective, these methods require extensive training data and may struggle with rapidly changing workloads. In contrast, our proposed LFRU policy adapts in real-time, leveraging observed correlations between client requests without relying on predictive models, making it more responsive to dynamic environments. Reinforcement learning (RL) methods, such as [15] and [16], optimize eviction decisions by adjusting to changing network conditions. However, these methods require continuous retraining to stay accurate. LFRU avoids this by inferring request-following relationships in real time, ensuring efficient eviction decisions despite fluctuating access patterns. The works most closely aligned with our approach are [17] and [18]. In [17], authors use a neural network model to model the inter-relationships between content requests, whereas LFRU achieves similar benefits with lower computational overhead. The Follow-The-Regularized-Leader-based approach in [18] optimizes cache decisions using historical data and regularization, but LFRU achieves comparable adaptability with a simpler, more direct inference mechanism.

Finally, hybrid strategies combining ML with traditional caching techniques, as in [19], aim to improve performance but often rely on complex models and incur higher computational costs. LFRU, by contrast, is a lightweight, observational approach, providing a more efficient alternative in environments where fast adaptation and low computational overhead are critical.

B. Paper contributions

In summary, we make the following key contributions. First, we introduce the *grouped client request model*, a generalization of the IRM that formally captures inter-client correlations (see Section II). Second, we derive a *working-set approximation* for LRU hit probabilities, demonstrating that LFU becomes suboptimal for large caches in correlated settings (see Section II). Third, to address these limitations, we propose *Least Following and Recently Used (LFRU)*, a lightweight online policy that dynamically adapts to structured request patterns (see Section III). Finally, using custom VR-based datasets, we empirically demonstrate that LFRU improves cache hit ratios by up to 2.9× over LRU and 1.9× over LFU (see Section IV).

II. System Model, analysis and simulation results

A. Model for cache

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

B. Model for grouped client request patterns

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

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

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

Definition 1 (Grouped Client Request Model)**.** *The leader of each group generates request for data objects independently of other requests, following a stationary Poisson process. Followers in each group make the same data object requests but with a possibly random delay (or advancement) relative to the time the leader made the request.*

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

$\sum_{d\in\mathcal{D}^g}\lambda^g(d)$. *The probability that the leader requests data object* $d\in\mathcal{D}^g$ *is denoted as* $p^g(d)=\frac{\lambda^g(d)}{\lambda^g}$.

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

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

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

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

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

Remark 1 (Extending the Grouped Client Request Model)**.** *While this model assumes that followers always request data after their leader, it can be easily extended to allow followers to randomly opt out of following the leader's requests. This is a potential direction for our future work.*

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

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

Quantifying $T_b(d)$ is important for determining the probability that d will be in the cache under the LRU policy. In the literature, two approximations simplify this calculation, and these approximations become more accurate as the total number of data objects, D, increases (i.e., when $D\gg 1$). For further details, see [7], [8].

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

Approximation 2: The dependence of $t_b(d)$ on the specific data object d can be neglected. This approximation is commonly used and justified in the literature [7], [8], particularly when the request probability $\sum_{g\in\mathcal{G}^d}p^g(d)$ is small relative to that of the remaining data objects request probabilities. This approximation becomes exact when the request probabilities are uniform.

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

$$b=\sum_{d\in\mathcal{D}}\mathbb{P}\left(\substack{d\text{ was requested at least once}\\ \text{in }[-t_b^*,0]}\right)s(d) \tag{1}$$

where the right hand side captures the size of set of data objects present in the cache at time 0 without loss of generality and since under LRU a data object stays in the cache for t_b^* amount of time after it is requested, we focus on the case whether a data object was requested at least once in the time interval $[-t_b^*,0]$. This is referred to as the working set approximation, which has been shown to be accurate as the number of data objects scale. We define an event $V_\tau^{l^g,i}$ as follows: the leader client l^g of group g makes a request at time τ, and its $i-$th follower's request for the same data object does not occur within $[-t_b^*,0]$. This condition is equivalent to $\{\tau+\Delta_i^g\notin[-t_b^*,0]\}$. Let $p(d,t_b^*)$ denote the probability that data object d is requested at least once in $[-t_b^*,0]$. We compute it in the following lemma.

Lemma 1 (Working Set Approximation)**.** *Consider a cache of size* b *that follows the LRU eviction policy and serves a grouped client request pattern (see Section II-B for details). The probability that a data object* d *is requested at least once within the time interval* $[-t_b^*,0]$ *is given by*

$$p(d,t_b^*)=1-e^{-\sum_{g\in\mathcal{G}^d}\int_{-\infty}^{\infty}\lambda_\tau^g(d,t_b^*)\,d\tau}, \tag{2}$$

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

$$q_\tau^g(t_b^*)=\begin{cases}1-\mathbb{P}\left(\bigcap_{i=1}^{f^g}V_\tau^{l^g,i}\right) & \text{if } \tau\in[-\infty,-t_b^*)\cup(0,\infty),\\ 1 & \text{if } \tau\in[-t_b^*,0].\end{cases} \tag{3}$$

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

$$b=\sum_{d\in\mathcal{D}}p(d,t_b^*)s(d). \tag{4}$$

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

The leader of group g generates requests for d as a PPP with rate $\lambda^g(d)$. Consider the leader of group g requests d at time τ, then its i-th follower requests it at time $\tau+\Delta_i^g$. A

request for data object d from a client in group g can fall in the interval $[-t_b^*, 0]$ either due to the leader's request occurring within the interval or due to a follower's request. Formally, we define $q_\tau^g(t_b^*)$ as the probability that at least one client in group g requests d in $[-t_b^*, 0]$, given that the leader made a request at time τ. This probability is given by

$$q_\tau^g(t_b^*) = \begin{cases} 1 - \mathbb{P}\left(\bigcap_{i=1}^{f^g} V_\tau^{l^g,i}\right) & \text{if } \tau \in [-\infty, -t_b^*) \cup (0, \infty), \\ 1 & \text{if } \tau \in [-t_b^*, 0]. \end{cases} \tag{5}$$

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

$$e^{-\int_{-\infty}^{\infty} \lambda_\tau^g(d, t_b^*)\, d\tau}. \tag{6}$$

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

$$p(d, t_b^*) = 1 - \mathbb{P}(\text{No client in } \mathcal{G}^d \text{ requests } d \text{ in } [-t_b^*, 0]), \tag{7}$$

$$= 1 - e^{-\sum_{g \in \mathcal{G}^d} \int_{-\infty}^{\infty} \lambda_\tau^g(d, t_b^*)\, d\tau}. \tag{8}$$

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

$$b = \sum_{d \in \mathcal{D}} p(d, t_b^*) s(d). \tag{9}$$

□

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

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

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

$$p^L(d, g) = 1 - [1 - p(d, t_b^*)] \left[\mathbb{P}\left(\bigcap_{i=1}^{f^g} E^{l^g,i}\right)\right]. \tag{10}$$

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

$$p^F(d, g; i) = 1 - [1 - p(d, t_b^*)]\, \mathbb{P}\left(\bigcap_{\substack{j=1 \\ j \neq i}}^{f^g} E^{i,j} \cap E^{i,l^g}\right). \tag{11}$$

Proof. To calculate the hit probability for a data object d requested by the leader l^g of group g, assume without loss of generality that this request occurs at time 0. The leader's hit probability depends on two factors: first, the randomly distributed request from l^g's followers around time 0: l^g can get a hit if at least one follower in group g requests data object d within $[-t_b^*, 0]$ since the followers can request for a data object before their leader does (see Section II-B for details), second, requests other than the randomly distributed request from l^g's followers around time 0: by Slivnyak's theorem [20], the remaining requests still follow a MPPP and thus l^g can get a hit if at least one client, from any group, requests data object d within the interval $[-t_b^*, 0]$ which is given by the earlier lemma under Approximations 1 and 2.

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

$$p^L(d, g) = 1 - \mathbb{P}\left(\substack{\text{No follower in } g \\ \text{requests } d \text{ in } [-t_b^*, 0]}\right) p(d, t_b^*). \tag{12}$$

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

$$\mathbb{P}\left(\substack{\text{No follower in } g \\ \text{requests } d \text{ in } [-t_b^*, 0]}\right) = \mathbb{P}\left(\bigcap_{i=1}^{f^g} E^{l^g,i}\right). \tag{13}$$

Now, consider the hit probability for a data object d requested by the i-th follower of group g. Again, assume without loss of generality that this request occurs at time 0. As before the follower's hit probability depends on two considerations: first, requests from clients other than i-th follower in group g: i-th follower can get a hit if at least one other client in group g requests data object d within $[-t_b^*, 0]$, second, requests other than the requests already considered: again using Slivnyak's theorem [20], the remaining requests still follow a MPPP and thus i-th follower can get a hit if at least one client (from any group) requests the data object d within the interval $[-t_b^*, 0]$ which is given by the earlier lemma under Approximations 1 and 2.

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

$$p^F(d, g; i) = 1 - \mathbb{P}\left(\substack{\text{No client except the } i\text{-th} \\ \text{follower in } g \text{ requests } d \text{ in } [-t_b^*, 0]}\right) [1 - p(d, t_b^*)] \tag{14}$$

From our earlier definition of events, we obtain:

$$\mathbb{P}\left(\substack{\text{No client except the } i\text{-th follower in} \\ g \text{ requests } d \text{ in } [-t_b^*, 0]}\right) = \mathbb{P}\left(\bigcap_{\substack{j=1 \\ j \neq i}}^{f^g} E^{i,j} \cap E^{i,l^g}\right). \tag{15}$$

□

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

D. Caching Policies

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

E. Performance Metric

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

$$H_{\mathbf{\Lambda},\pi} = \frac{1}{\sum_{g\in\mathcal{G}} \lambda^g(1+f^g)} \Bigg(\sum_{g\in\mathcal{G}} \sum_{d\in\mathcal{D}^g} \lambda^g(d) h^{g,l^g}_{\mathbf{\Lambda},\pi}(d) + \sum_{g\in\mathcal{G}} \sum_{i=1}^{f^g} \sum_{d\in\mathcal{D}^g} \lambda^g(d) h^{g,i}_{\mathbf{\Lambda},\pi}(d) \Bigg). \tag{16}$$

F. An optimal (offline) static caching policy

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

$$\underset{\mathbf{x}}{\text{maximize}} \quad \sum_{g\in\mathcal{G}} \sum_{d\in\mathcal{D}^g} \lambda^g(d) x(d)(1+f^g) \tag{17a}$$

$$\text{subject to} \quad \sum_{d\in\mathcal{D}} x(d)s(d) \le b, \tag{17b}$$

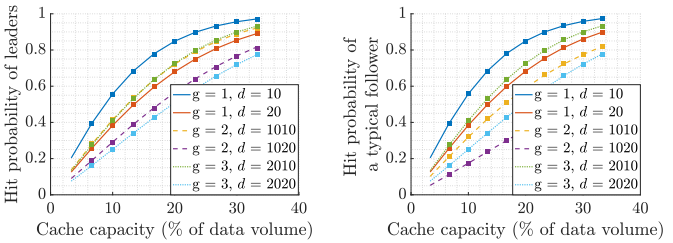
$$x(d) \in \{0,1\}, \forall\ d \in \mathcal{D} \tag{17c}$$

where constraint Eq. 17b ensures that the cache size does not exceed the capacity b.

G. Simulation results

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

1) How accurate is the working set approximation for grouped client request pattern under LRU?: **Setup.** We consider a caching system with three groups ($G = 3$), where each group requests data objects from a distinct set. The data object sets for the groups are defined as $\mathcal{D}^1 = \{1, 2, \ldots, 1000\}$, $\mathcal{D}^2 = \{1001, 1002, \ldots, 2000\}$, and $\mathcal{D}^3 = \{2001, 2002, \ldots, 3000\}$. Each group consists of one leader and several followers, with the number of followers in each group being $f^1 = 6$, $f^2 = 4$, and $f^3 = 3$. The delay between the i-th follower (for all $i \in \{1, \cdots, f^g\}$) and the leader of group g is modeled as independent and identically distributed (i.i.d.) random samples from a uniform distribution: for group 1, $\Delta^1_i \sim \mathrm{U}[-10, 20]$, for group 2, $\Delta^2_i \sim \mathrm{U}[15, 30]$, and for group 3, $\Delta^3_i \sim \mathrm{U}[-5, 40]$. The request probability $p^g(d)$ for an object d in group g follows a Zipf distribution with parameter 1, identical across all groups. Requests from group leaders arrive according to a Poisson process, with the following rates: $\lambda^1 = 10$, $\lambda^2 = 8$, and $\lambda^3 = 12$. Additionally, object sizes are assigned such that even-numbered objects have size 2 and odd-numbered objects have size 5, uniformly across all groups.



(a) Hit probability of leaders.

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

Fig. 2: Hit probability against cache capacity for different clients and data objects under LRU.

Results Discussion. Fig. 2 shows the probability of hit for various objects requested by clients (leaders and followers) in different groups under LRU across various cache capacities, expressed as a percentage of the total data volume (computed as the number of data objects multiplied by their average size). Since follower delays in each group are i.i.d., the hit probability for all followers in a group is identical. The squares in the figure represent simulation results for the LRU policy, obtained from long simulation runs to ensure reliable accuracy. The lines are derived from the working set approximation for LRU. As shown, the approximation aligns almost perfectly with the simulation results across all clients, demonstrating its high accuracy for practical applications.

III. Caching Based on Inferred Temporal Dependence Relations

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

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

A. Notation

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

$$c(d,t) = \begin{cases} \operatorname{argmax}_c \left\{ \substack{a_n \cdot \mathbf{1}(d_n = d) \cdot \mathbf{1}(a_n < t) \\ :c \in \mathcal{C},\, n \in \mathbb{Z}^+} \right\} & \text{if client exists,} \\ -1 & \text{otherwise.} \end{cases} \tag{18}$$

where $\mathbf{1}(\cdot)$ is the indicator function. If no such client exists, the function returns -1. Next, we define $n^c(t)$ as the index of the last request made by client c at or before time t:

$$n^c(t) = \operatorname*{argmax}_n \left\{ a_n^c \leq t : n \in \mathbb{Z}^+ \right\}. \tag{19}$$

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

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

$$f_n^{c_1,c_2} = \mathbf{1}(d_n^{c_2} \in M(a_n^{c_2}))\mathbf{1}(c(d_n^{c_2}, a_n^{c_2}) = c_1) \tag{20}$$

B. Caching policy

1) Least Following and Recently Used (LFRU (w)): The LFRU policy can improve on traditional LRU caching policy by considering inferred temporal relationships (following events) between clients' requests. This policy manages cache evictions using both the recency of a request and how often other clients have followed the client who made the request. The goal is to keep data objects that were recently accessed and are more likely to be accessed again soon, based on these past patterns of client requests.

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

$$F^{c_1,c_2}(t) = \begin{cases} \sum_{n=n^{c_2}(t)-w}^{n^{c_2}(t)} f_n^{c_1,c_2} & c_1 \neq c_2 \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

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

Description of the Policy: When a new request (a_m, d_m, c_m) arrives, we first check whether it results in a cache hit or a cache miss. If it is a cache hit, the requested object is moved to the most recent position in the access order to reflect its recency. If it is a cache miss, the object is added to the cache as the most recent entry, and evictions are performed as required. To determine which object(s) to evict at time a_m, we first update the matrix $F(a_m)$, which captures the number of following events between pairs of clients. Next, using this matrix, we calculate the maximum number of times that other clients have followed each client's requests. Finally, objects associated with clients having the fewest following events are evicted first. If multiple objects correspond to clients with the same following event count, least recently accessed object is evicted. Note if no following events have been seen then the caching policy corresponds to LRU.

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

$$F^{c_1,c_2}(t) = \begin{cases} \left\lfloor \sum_{n=n^{c_2}(t)-w}^{n^{c_2}(t)} \gamma^{(n^{c_2}(t)-n)} f_n^{c_1,c_2} \right\rfloor & c_1 \neq c_2 \\ 0 & \text{otherwise.} \end{cases} \tag{22}$$

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

IV. Simulation results

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

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

Caching policies. We consider two additional caching policies for evaluation. **Belady**'s algorithm evicts the data object requested furthest in the future, serving as an optimal but non-causal baseline that requires full knowledge of the request sequence. We also evaluate **Sieve** [21], a state-of-the-art policy that retains recently accessed objects using a single queue and a "hand" pointer to evict the least recently visited object with its visited bit unset. Both policies are evaluated on traces where all data objects have the same size.

A. Simulations for the grouped client request model

Setup. We consider a caching system with three groups ($G = 3$), where each group requests data objects

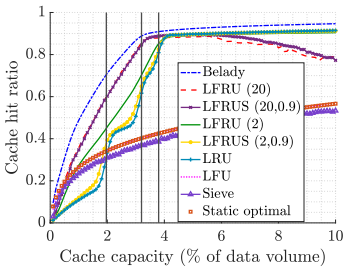


Fig. 3: Cache hit ratio against cache capacity under different caching policies for grouped client request model.

Fig. 4: The virtual city of the simulation.

from a distinct set: $\mathcal{D}^1 = \{1, 2, \ldots, 1000\}$, $\mathcal{D}^2 = \{1001, 1002, \ldots, 2000\}$, and $\mathcal{D}^3 = \{2001, 2002, \ldots, 3000\}$. Each group consists of one leader and several followers, with the number of followers given by $f^1 = 8$, $f^2 = 6$, and $f^3 = 4$. The delay between the i-th follower ($i \in \{1, \ldots, f^g\}$) and the leader of group g is modeled as a constant and ordered, with values $\Delta_i^1 = 10i$, $\Delta_i^2 = 20i$, and $\Delta_i^3 = 30i$ for Groups 1, 2, and 3, respectively. The request probability $p^g(d)$ for an object d in group g follows a Zipf distribution with parameters 0.8, 0.85, and 0.9 for Groups 1, 2, and 3, respectively. Requests from group leaders arrive according to a Poisson process with rates $\lambda^1 = 10$, $\lambda^2 = 15$, and $\lambda^3 = 20$. The size of all data objects is set to 1.

Results Discussion. Fig. 3 presents the cache hit ratio for different caching policies across various cache capacities, expressed as a percentage of the total data volume (computed as the number of data objects multiplied by their average size). The cache capacity ranges from small (0.1%) to large (10%). As expected, Belady's policy, which has full knowledge of future requests, performs the best.

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

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

Under the grouped client request model, LRU performs well when the characteristic time, t_b^*—the time before a newly introduced object is evicted if there are no additional requests for this object—is longer than the delay between follower requests. In such cases, every follower request results in a cache hit, regardless of the requested object. To illustrate this effect, vertical lines in Fig. 3 mark cache capacities of 2%, 3.2%, and 3.8% of the total data volume. These correspond to cache sizes where the characteristic time matches follower delays in Groups 1, 2, and 3, with delays of 10, 20, and 30 units, respectively. As expected, around these points, the LRU hit rate exhibits a sharp increase.

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

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

B. Simulations for emulated VR request traces

1) Simulation environment for generating cache request traces.: We generate cache request traces using a simulator that models a virtual desert city covering an area of 376.29×608.22 m^2, populated with 1,176 objects such as buildings, trees, and fountains. In the simulation, a virtual reality (VR) client moves through the city at a height of 1 to 2 meters above ground—typical for VR headset usage while

sitting or standing. Since buildings can block the view, clients can only request data for objects currently visible from their position.

To determine visibility, we dynamically compute the set of objects visible from all directions around the client's current location. This method is more effective than assuming a fixed or random view direction, as it ensures the client receives data for all possible directions without having to specify one. This is especially important because visibility computation is expensive, and clients can quickly change their viewpoint. Relying on a single view direction would lead to inefficiencies and delays.

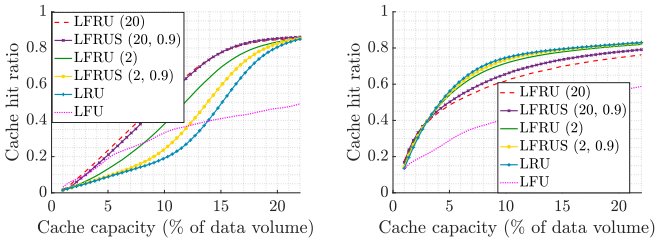
Clients move along pre-defined, looped paths through the city's alleys, avoiding collisions with objects, as shown in Fig. 4. There are always three client groups in the simulation, each with five clients. Each group follows a distinct path, with one leader and four followers moving in the same direction. All paths are bidirectional, non-intersecting, and clients move at a constant speed of 2 m/s.

The simulation progresses in discrete time slots. In each slot, clients update their position based on elapsed time and recompute visibility. A client sends a request for an object if it becomes visible in the current slot but was not visible in the previous one. We assume the client's cache can store all requested data from one slot until the next.

The size of each requested object depends on its pixel resolution, texture, geometric complexity, and the client's distance from it. Each object has three versions, allowing a tradeoff between size and visual quality. If a client is within 10 m of an object, it requests the highest-quality version; beyond 50 m, it requests the lowest-quality version; and between 10 m and 50 m, it requests the medium version. For example, an object's high-, medium-, and low-quality versions may be 1 MB, 0.5 MB, and 0.1 MB, respectively. Each request must be served with the exact version selected by the client, as versions are not interchangeable.

2) Trace 1: Unstructured follower requests.: **Setup.** In this trace, all clients within the same group are initialized near a common point on their designated path (Fig. 1b). Each client starts at a random position within 4 m of this point. Clients do not enter buildings during navigation. All followers move in the same direction as their group's leader, maintaining a constant delay of 0.5 seconds. Specifically, if the leader changes direction at a corner, all followers adjust their movement 0.5 seconds later. Each client maintains a local cache of previously requested objects managed using the LRU policy. The size of each local cache is set to 5% of b, where b is the total cache capacity of the edge or cloud server. When a client makes a request, it first checks its local cache. If the requested item is not found (a cache miss), the client forwards the request to the main cache (edge or cloud server). We have considered a local cache for each client as described for all simulation results discussed hereafter.

Results discussion. Fig. 5b shows the cache hit ratio for different caching policies across a range of edge/cloud cache capacities. These capacities are expressed as a percentage



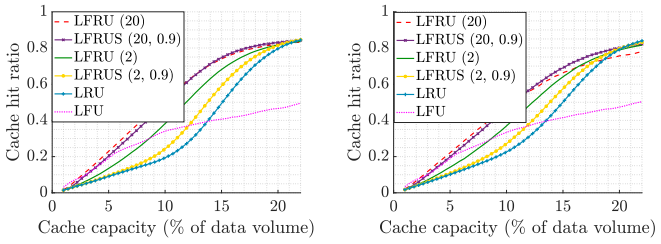
(a) Structured following. (b) Unstructured following.

Fig. 5: Cache hit ratio against cache capacity under different caching policies for different correlation settings.

of the total data volume, calculated as the number of data objects times their average size, and range from 1% (small) to 22% (large). Our results show that LRU performs best when the cache capacity is moderate to large. This is because, in unstructured following, followers are spread around the leader in a staggered way and do not consistently request the same data objects. As a result, fewer consistent following events occur, making inferred temporal relationships less useful for caching. In such cases, recency-based eviction (as in LRU) outperforms policies that rely on inferred causal patterns. This is supported by the performance gap between LFRUS and LFRU with the same window size—smoothing helps by giving more weight to recent following events. Smaller window sizes also help the policy adapt quickly by removing outdated following patterns. For small cache capacities, LFRU and LFRUS work better with larger window sizes. Instead of tracking the recency of all clients, these policies focus on nearby clients with more persistent following behavior, leading to better cache hit ratios.

3) Trace 2: Static following (Structured follower requests): **Setup.** In this trace, all followers move along the path by following their leader (Fig. 1a). The leader is initialized at a random point on the path, and each follower is subsequently initialized with a constant gap/distance of 4 m. For example, if the leader starts at position x, the first follower is placed at $x-4$ m, the second at $x-8$ m, and so on, assuming movement in the positive direction. Throughout the simulation, all clients maintain their initial relative spacing along the path.

Results discussion. Fig. 5a presents the cache hit ratio for different caching policies across various cache capacities, ranging from small (1%) to large (22%). Our results indicate that LFRU/LFRUS with $w = 20$ consistently outperforms all other caching policies across all cache capacities. This is because, in structured following, followers sequentially request data after their leader and tend to repeat the leader's requests. Once these following events are detected, they enhance cache eviction decisions by prioritizing the retention of objects requested by a client (excluding the last follower) within a group, as these objects are likely to be requested again by another follower in the same group. The benefit of retaining detected following events longer is reflected in the performance gap between LFRU with $w = 20$ and $w = 2$. A larger window en-



(a) p = 150 seconds. (b) p = 30 seconds.

Fig. 6: Cache hit ratio against cache capacity under different caching policies for Periodic order shuffling.

ables the policy to better exploit persistent following patterns, improving cache hit ratios. In this setting, smoothing becomes unnecessary, as following behavior remains stable over time.

4) Trace 3: Periodic order shuffling: **Setup.** In this trace, the setup remains the same as in Trace 2, except that followers periodically swap positions in pairs. Every p seconds (30s or 150s), adjacent followers swap positions: the first follower swaps with the second, and the third with the fourth. The swap occurs gradually over a duration of 3 seconds, rather than instantaneously. During this process, clients continue their normal movement along the path, but their relative distances temporarily change as they transition to their new positions.

Results discussion. Figures 6a and 6b show the cache hit ratio for different caching policies when the period p is large and small, respectively. The period p indicates how long a specific following behavior lasts. When p is large, policies that use inferred temporal dependence patterns perform well, with only a minor drop compared to Fig. 5a, which is negligible. When p is small, LFRU with $w = 20$ performs worse at larger cache sizes than it does with a larger p. This happens because follower order changes more often, causing the policy to misidentify following relationships. As a result, the cache may evict objects tied to clients who are no longer being followed, which hurts performance. Using a smaller window, like in LFRU with $w = 2$, helps the policy respond faster by forgetting outdated following events. Another approach is to assign weights to following events so that only persistent patterns influence decisions, as in the LFRUS variants. For small cache sizes, a larger window is still helpful since LFRU may not detect all following patterns. In these cases, even with changing follower order, the performance remains stable.

V. Conclusion

Collaborative VR and social-metaverse applications can introduce edge traffic patterns where client data requests are shaped by group movement rather than isolated actions. In this paper, we introduced the *Grouped Client Request Model* to formally capture these inter-client dependencies. Our analysis demonstrated that standard policies like LRU fail to account for the social correlations inherent in collaborative VR, leading to suboptimal performance. To address this, we proposed *Least Following and Recently Used (LFRU)*, an adaptive policy that dynamically infers the group-following behavior associated with future content demand. By prioritizing objects based on group movement structure, LFRU enhances eviction decisions. Our evaluations on VR datasets show LFRU achieves up to **2.9**× and **1.9**× improvements in cache hit ratio over LRU and LFU, respectively. These findings highlight the necessity of *social-aware resource management* to support the scalable, low-latency experiences required by future shared virtual worlds.

Acknowledgment

This work was supported by National Science Foundation CNS-2212202 Award.

References

[1] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *Proc. ACM SIGMETRICS*, 1990, pp. 143–152.

[2] P. R. Jelenković, "Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities," *The Annals of Applied Probability*, vol. 9, no. 2, pp. 430–464, 1999.

[3] G. Hasslinger, M. Okhovatzadeh, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "An overview of analysis methods and evaluation results for caching strategies," *Comput. Netw.*, vol. 228, p. 109583, 2023.

[4] W. King, "Analysis of paging algorithms," in *Proc. IFIP Congress*, 1971, pp. 485–490.

[5] P. J. Denning and S. C. Schwartz, "Properties of the working-set model," *Commun. ACM*, vol. 15, no. 3, pp. 191–198, 1972.

[6] R. Fagin, "Asymptotic miss ratios over independent references," *J. Comput. Syst. Sci.*, vol. 14, no. 2, pp. 222–250, 1977.

[7] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: modeling, design and experimental results," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 7, pp. 1305–1314, 2002.

[8] C. Fricker, P. Robert, and J. Roberts, "A Versatile and Accurate Approximation for LRU Cache Performance," in *Proc. ITC*, 2012.

[9] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, 1971.

[10] P. R. Jelenković and A. Radovanović, "Asymptotic optimality of the static frequency caching in the presence of correlated requests," *Oper. Res. Lett.*, vol. 37, no. 5, pp. 307–311, 2009.

[11] P. Jelenkovic and A. Radovanovic, "Least-recently-used caching with dependent requests," *Theor. Comput. Sci.*, vol. 326, pp. 293–327, 2004.

[12] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "DeepCache: A deep learning based framework for content caching," in *Proc. NetAI*, 2018, pp. 48–53.

[13] ——, "Making content caching policies 'smart' using the deepcache framework," *SIGCOMM Comput. Commun. Rev.*, vol. 48, no. 5, pp. 64–69, 2019.

[14] Q. Fan, X. Li, J. Li, Q. He, K. Wang, and J. Wen, "PA-Cache: Evolving learning-based popularity-aware content caching in edge networks," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 2, pp. 1746–1757, 2021.

[15] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman, "RL-Cache: Learning-based cache admission for content delivery," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2372–2385, 2020.

[16] L. Lai, F.-C. Zheng, W. Wen, J. Luo, and G. Li, "Dynamic content caching based on actor-critic reinforcement learning for IoT systems," in *Proc. IEEE VTC-Fall*, 2022, pp. 1–6.

[17] Y. Im, P. Prahladan, T. H. Kim, Y. G. Hong, and S. Ha, "SNN-cache: A practical machine learning-based caching system utilizing the inter-relationships of requests," in *Proc. CISS*, 2018, pp. 1–6.

[18] F. Faticanti and G. Neglia, "Optimistic online caching for batched requests," arXiv:2310.01309, 2023.

[19] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *J. ACM*, vol. 68, no. 4, 2021.

[20] F. Baccelli and B. Blaszczyszyn, *Stochastic Geometry and Wireless Networks: Volume I Theory*, 2010.

[21] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, "SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches," in *Proc. USENIX NSDI*, 2024, pp. 1229–1246.