Edge Cache on WiFi Access Points: Millisecond-Level App Latency Almost for Free

Zhengquan Li, Summit Shrestha, Zheng Song
Department of Computer and Information Science
University of Michigan at Dearborn
Dearborn, Michigan
{zqli, summitsh, zhesong}@umich.edu

Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
tilevich@cs.vt.edu

***Abstract*—To achieve low execution latency, time-sensitive applications, including AR/VR and autonomous driving, cache data at the edge of the network, close to end users. However, existing edge caches often fail to deliver low latency due to the inefficiency of DNS requests and the physical remoteness of their users. The solution described herein addresses these inefficiencies by presenting a millisecond-level, lightweight caching architecture that operates directly on widely deployed WiFi access points (APs). Specifically, our architecture interposes another level of caching closer to the client and is fine-tuned for APs's limited cache memory. Our solution (1) features a novel algorithm for managing cache at the AP level; (2) allows the cache query workflow to proceed at full speed; and (3) requires no changes to the application logic. Our evaluation demonstrates that our reference implementation can decrease application-level latency by as much as 76% compared to the existing solutions, without impacting AP core functions. Our caching architecture effectively improves application responsiveness by tapping into existing networking infrastructure, thus offering a powerful and cost-efficient system component for building emerging time-sensitive applications at the edge.**

***Index Terms*—Edge Caching, WiFi AP, Latency, DNS**

I. Introduction

To ensure the required QoS in delivering critical data and functionalities, mobile apps have traditionally relied on the cloud, whether they run on smartphones, IoT devices, or smart vehicles. Certain applications must fulfill stringent requirements of responding to the user's inputs in less than 20∼50 ms [1]–[3], a metric that we refer to as *application-level latency*. Examples include virtual reality (VR) gaming [4], remote surgical guidance [5], and autonomous driving [6]. Because traditional cloud setups are often incapable of meeting such stringent latency requirements, various edge-based caching solutions have been introduced. To reduce application-level latency, these solutions cache data at the edge of the network, close to end users.

However, recent studies reveal that existing edge caching solutions often fall short of meeting the latency requirements of many time-sensitive applications [7], [8]. These solutions typically follow a two-step workflow: (1) locate the nearest cache server via DNS resolution; (2) retrieve the cached data from it. As our investigation identifies, this workflow suffers from inherent inefficiencies. First, DNS resolution is notoriously slow and can take as much as 20 ms to fully complete. Also, with cache servers located as far away as 14 hop from the end user, the average RTT can exceed 30 ms.

Motivated by these observations, we introduce a novel system caching architecture that utilizes widely deployed WiFi Access Points (AP). WiFi APs are a fundamental component of the modern Internet infrastructure, ubiquitously deployed one hop away from the users. Our architecture takes advantage of the low RTT between APs and their connected users to provide **millisecond-level** caching.

However, adding custom caching to WiFi APs, without degrading their core functions, requires special care due to APs' limited physical cache memory. To that end, our architecture is multilayered: we interpose an AP-based cache as an intermediate layer between the user and the edge-based cache. This way, the user first queries the AP-based cache, and only upon a cache miss, queries the edge-based cache. This setup makes it possible to efficiently utilize the limited AP cache memory for data with the highest latency impact, thus providing a **lightweight** caching solution. We dubbed the reference implementation of our caching architecture as APE-Cache (AP and Edge Cache).

Among the novel features of APE-Cache are: 1) An algorithm for managing cache that strategically prioritizes cacheable objects based on how they impact the application-level latency, optimizing the utilization of limited AP cache memory; 2) A middleware system that efficiently integrates AP-based cache lookup into the DNS query process of edge-based caching, thereby minimizing the overhead of cache misses; and 3) An intuitive programming model for specifying the data to cache, without modifying application logic.

By describing the design, implementation, and evaluation of APE-Cache, this paper makes the following contributions:

- We highlight the problem of high latency in existing edge caching systems, motivating the need for more efficient solutions for caching at the edge.
- We introduce the design and implementation of APE-Cache, a millisecond-level and lightweight caching solution that leverages WiFi APs. Its superior performance is due to its low object retrieval latency and high cache hit ratio for objects crucial to application-level latency. APE-Cache offers an intuitive programming model and is amenable to integration with off-the-shelf APs.

- Our evaluation reveals that APE-CACHE significantly reduces the application-level latency of realistic mobile apps by as much as 76%, compared with existing edge-based caching.

The rest of this paper is organized as follows: Section II provides the background and motivation of this research; Section III highlights the technical challenges of AP caching; Sections IV and V explain the design and evaluation of APE-CACHE, respectively; Section VI discusses related work, and Section VII presents concluding remarks.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce background on edge-based caching, followed by an empirical study that reveals the performance issues with state-of-the-practice edge caches. We then study the feasibility of caching on APs.

A. Edge-based Caching Workflow

Edge caching systems are crucial for enhancing the efficiency of data delivery, and can be broadly categorized into Content Delivery Networks (CDN) and Cache-equipped Base-Stations (CBS). Both categories share a similar workflow that is transparent to software developers [9], [10]: 1) when programming an app, developers specify the URLs of remote data objects; 2) at runtime, the DNS system parses the domain names of cacheable objects to the IP addresses of the nearest cache servers; 3) the app sends HTTP requests to the nearest cache servers to retrieve the cacheable objects.

To demonstrate the workflow, we use Akamai [1], the most widely deployed CDN-based caching system. When an app requests "`www.apple.com/image.jpg`", deployed on the Akamai caching system, it first looks up the cache server by sending DNS requests and then retrieves the cached image. This process ensures that users receive requested data swiftly and efficiently, bypassing the need to connect to distant origin servers, as shown in Fig. 1.

Cache Lookup: 1) The user's runtime initiates the process by sending a DNS request to the local DNS (LDNS) to resolve the domain name `www.apple.com`; 2) The LDNS checks its cache for the IP address of `www.apple.com`. If hit, the LDNS returns the cached IP to the user. Otherwise, it sends a resolution request to Apple's authoritative DNS; 3) The authoritative DNS (ADNS) resolves the domain name, which points to an Akamai server: `www.apple.com.edgekey.net` (a CNAME record of the domain name). The ADNS subsequently returns this CNAME record to the LDNS; 4) The LDNS forwards the CNAME request to the Akamai's DNS service, which further identifies and provides the IP address of the nearest caching server to the LDNS; 6) The LDNS then relays this IP address back to the user.

Cache Retrieval: 7) Using this IP address, the user issues an access request to the caching server; 8) The caching server responds with the requested data: a) If the data is already cached, it directly sends it to the user; b) If the data is uncached, the server retrieves it from the origin server before sending it to the user.

[1]https://www.akamai.com/

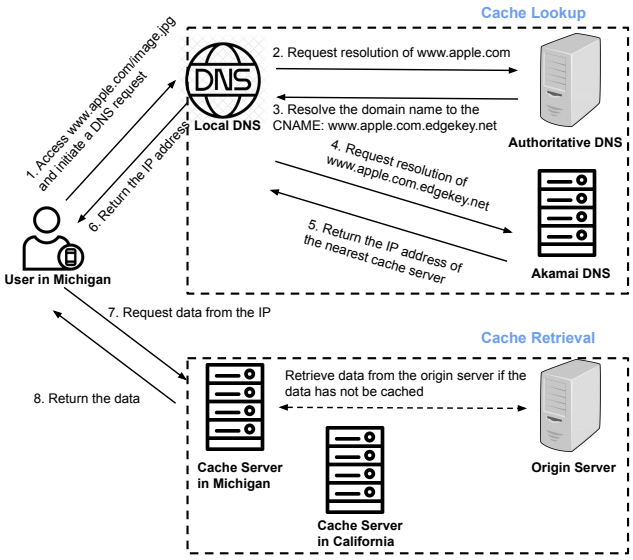


Fig. 1: The Workflow of Fetching Data from Edge Caching

B. Performance Issues with Edge Caching Systems

For edge-based caching, latency is crucial for user experience. To understand the performance of the state-of-the-practice edge caching systems and their bottleneck, we conducted an empirical study that requested cached data from Akamai's caching system from various locations. Following the two-stage workflow, we first directly measured the latency for DNS resolution of cache lookup. Then we indirectly measured the latency for cache retrieval as the Round-trip time (RTT) and hops count between the client and the redirected caching server.

We chose the three most widely visited websites using Akamai, including www.apple.com, www.microsoft.com, and www.yahoo.com, and sent requests from three locations distributed globally, including Michigan (USA), Tokyo (Japan), and São Paulo (Brazil). For this purpose, we developed a Python tool that leverages the `socket.gethostbyname` library for DNS processing, converting hostnames to the IP addresses of caching servers. We also employed the Linux `Ping` command to measure the RTT between the client and the resolved IP address, and `traceroute` to determine the network hops from the client to the IP address. To ensure precise measurements of DNS resolution latency and RTT, we executed the tool 100 times, and Table I summarizes the average latency of DNS resolution, RTT, and network hops between the client and the resolved IP addresses.

We observe that: 1) The average latency involved in DNS resolution for pinpointing the caching server is 22ms; 2) The average Round-Trip Time (RTT) between the client and the caching server stands at 38ms, encompassing 14 network hops in one direction; 3) The distribution of caching servers is not universal, as evidenced by the absence of an available

Location	Metric	Apple	Microsoft	Yahoo
Michigan, US	DNS Resolution (ms)	18	19	21
	RTT (ms)	34	33	53
	Number of Hops	13	13	16
Tokyo, Japan	DNS Resolution (ms)	18	26	27
	RTT (ms)	22	27	93
	Number of Hops	7	10	13
São Paulo, Brazil	DNS Resolution (ms)	20	26	226
	RTT (ms)	19	19	156
	Number of Hops	12	10	15

TABLE I: Performance Measurement of Akamai Caching

Akamai cache server for Yahoo users in São Paulo, Brazil. Consequently, users in this region must retrieve data from the origin server, leading to significantly higher latency.

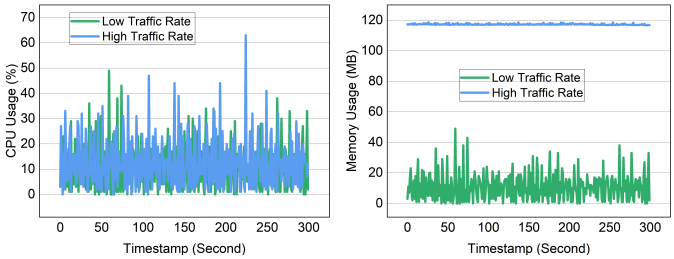
C. CPU/Memory of WiFi APs Available for Caching

The core concept of edge caching is reducing caching retrieval latency by deploying cache servers close to end users. To this end, WiFi APs are one hop away from the end users and could potentially further reduce latency. Hence, we explored whether the APs have enough CPU and memory resources for caching. We replayed pre-captured WiFi network traffic to an off-the-shelf WiFi router (i.e., GL-MT1300 powered by MT7621A CPU @ 880MHz with 256MB Memory). The traffic datasets consisted of two network flows that were captured in different network traffic rates, as showed in Table II. We used `Tcpreplay` to replay the two network flows and recorded the CPU and memory utilization of the router while replaying.

	Low Traffic Rate	High Traffic Rate
Size	9.4 MB	368 MB
Packets	14261	791615
Flows	1209	40686
Average packet size	646 bytes	449 bytes
Duration	5 minutes	5 minutes
Number of apps	28	132

TABLE II: Statistics of Public WiFi Traffic Datasets [11]

From Fig. 2 we observe that: 1) higher traffic rates significantly increase the load on the router, particularly impacting memory usage, which consistently hovers around 120MB; 2) even under high traffic conditions, there are still available CPU and memory resources on the router. The CPU usage remains well below 50%, and memory usage does not exceed half of the total capacity (i.e., 256MB).



(a) CPU usage (b) Memory usage

Fig. 2: CPU/Memory Usage of WiFi Router

To determine if these findings are indicative of a wider trend, we searched on Amazon using the keyword "WiFi router," and manually inspected the specifications (CPU frequency, RAM) of 22 products from the first page of results. We found all 15 routers over the price of $60 are equipped with similar or better CPU and RAM specifications than the one we tested.

To summarize, caching on APs is feasible thanks to the under-utilization of computational and storage resources on these devices. However, considering the modest hardware capabilities of WiFi APs and the necessity to allocate sufficient memory for critical functions such as routing and DNS resolution, the space available for caching on APs is significantly smaller compared to that in edge-based caching systems.

III. APE-Cache: Key Ideas and Considerations

This section introduces the key ideas and considerations behind APE-Cache. To accommodate the unique constraints of caching at AP, the key system design idea behind APE-Cache is to interpose AP-based cache as an intermediate layer between the user and the existing edge-based cache, so high-priority data objects can be cached on APs while the rest of them at the edges. By *priority* of data objects, we mean their impacts on the app-level latency and explain it using the following example.

A. Motivating Example

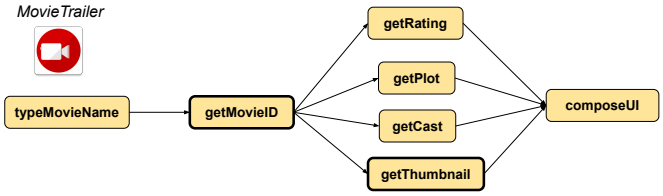


Fig. 3: The Logic of MovieTrailer app

Consider *MovieTrailer*, a real-world app that is open-sourced on Github [12] and released on Google Play. The main functionality of this app is to provide detailed movie information given the movie's name. To execute this functionality, the app first issues a request to convert the movie's name to a unique *movieID*. Then, the app issues four concurrent requests to retrieve the detailed information about the *rating*, *plot*, *cast*, and *thumbnail* of the movie identified by the given *movieID*. Finally, based on the four fetched data objects, the app composes and renders the UI, presenting the requested movie information to the user.

Notice that these data objects (i.e., movie ID, rating, plot, cast, and thumbnail) can be cached to shorten data fetching time, thereby reducing app-level latency. However, there is a high variance in how the fetching of each object contributes to app-level latency, with objects on the critical path of the app, i.e., the longest path (in duration) from start to finish, making a larger impact. In our example, the critical path is *getMovieID* → *getThumbnail*, as the *thumbnail* object generally requires more time to fetch due to its larger size compared to the other three peers (i.e., rating, plot, cast). Therefore, *movieID* and

thumbnail are considered to be of high priority. To minimize app-level latency within the physical limitations of cache memory space on AP, APE-CACHE should cache *movieID* and *thumbnail* on AP, while the remaining data objects on traditional edge caches.

B. Technical Issues and Solutions

The differences between caching on APs and edge servers, along with the need to combine their usage, pose the following main technical issues.

1) *How to Optimize Priority-aware Caching on APs:* Unlike traditional edge caches, which manage cache on the server based on data access frequency, priority-based caching on APs must be aware of app logic, data size, and real-time retrieval latencies, necessitating a more context-aware and sophisticated approach.
   Solution: APE-CACHE empowers developers to specify data priority in the source code. It also features a priority-aware cache management algorithm (PACM), detailed in Section IV-C.
2) *How to Reduce Cache Lookup Overhead:* Because it is unknown whether a data object is cached on AP or not, to retrieve a cacheable data object, an app follows a two-step lookup: (1) look up its connected AP, and, if missed, (2) switch to the edge-based cache system. The limited AP's memory cache space may cause a high cache miss rate, thus requiring APE-CACHE's lookup to be latency efficient to minimize the overall data retrieval latency.
   Solution: To locate the nearest edge server for a data object, clients first send DNS requests to their connected APs. APE-CACHE leverages this process to simultaneously initiate AP-based cache lookups, effectively minimizing overhead and potentially accelerating retrieval. Section IV-B describes the details.
3) *How to Reduce Programming Effort:* Existing edge caching systems are transparent to app developers. Developers only need to specify the URLs of cacheable objects, and can dynamically access the objects cached on the nearest edges by DNS queries. The same programming model cannot support caching on APs that only have IP addresses but not domain names.
   Solution: APE-CACHE provides an intuitive programming model for app developers. Developers can use Java annotations to configure APE-CACHE's runtime which objects should be further cached on APs. Section IV-A introduces our programming model in detail.

C. APE-CACHE Workflow

To understand APE-CACHE's workflow, consider Fig. 4, which also makes use of our earlier example app. As shown in the left side, the developer annotates cacheable objects and specifies their URLs, priority, and TTL (i.e., time-to-live). We define priority as positive integers, with higher values corresponding to greater priority. When the client requests a cacheable object (for example, fetching movieID by calling

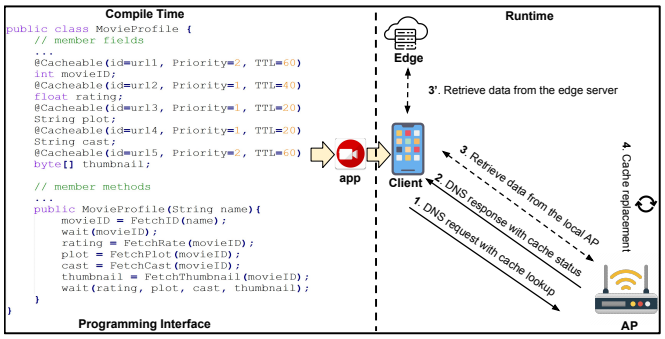


Fig. 4: APE-CACHE System Workflow

FetchID(name)), it sends a modified DNS request to its associated AP (step 1). The AP sends back the IP address of the edge server, as well as whether it can retrieve the *movieID* for a given movie name from the AP (step 2). The client will either retrieve data from the AP (step 3), or retrieve the data from the edge server using the IP address it receives (step 3'). For step 3, the AP will either return the cached data if available, or delegate the client's request to the server and return the obtained results. It also runs a priority-aware cache replacement algorithm to replace outdated data with newly received data (step 4).

IV. APE-CACHE System Design and Implementation

APE-CACHE comprises two subsystems, running on mobile clients and APs. On mobile clients, APE-CACHE enhances existing HTTP client libraries, such as OkHttp, Apache Http Client, and Retrofit. Their enhanced versions are then incorporated into mobile apps. The enhancements fall into two primary modules: programming support (detailed in Section IV-A) and cache lookup/fetching (Section IV-B). At runtime, the mobile client's cache lookup/fetching module collaborates with its AP counterpart. Collaboratively they first determine if a cacheable object is located on AP, and subsequently retrieve it. A dedicated cache management module on AP executes the cache replacement algorithm described in Section IV-C.

A. Programming Model

APE-CACHE offers a declarative programming model, in which *cacheable* objects can be marked by the `Cacheable` Java annotation, depicted in Fig. 6. In this model, developers annotate Java class fields as `Cacheable`, assigning their URL, priority, and Time-To-Live (TTL) attributes. For a concrete example see Fig. 4.

The `id` field uniquely identifies cacheable objects by representing their basic URLs without parameters. The `priority` field accepts values of 1 or 2, which stand for low and high priority, respectively. The `TTL` field indicates the duration in minutes for which the cacheable objects remain valid.

We process all `Cacheable` annotations to retrieve the cacheable objects. Subsequently, this client library intercepts each outgoing HTTP request to verify if its basic URL matches any of the cacheable objects. The matched requests are then directed to the cache-lookup and fetching module, bypassing the HTTP client library's standard processing.

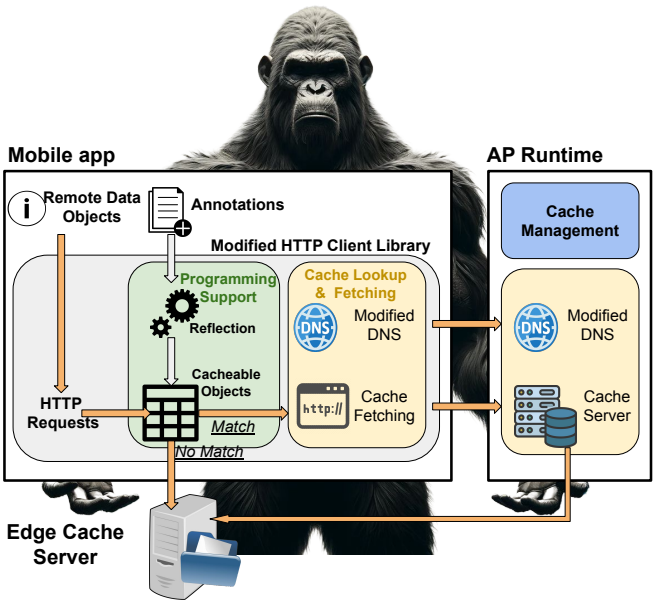


Fig. 5: The Overview of APE-CACHE

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.FIELD)
3 public @interface Cacheable {
4     String id();
5     int Priority();
6     int TTL();
7 }
```

Fig. 6: Cacheable Annotation

B. Cache Lookup and Fetching

Fig.7 illustrates the cache lookup and fetching workflows followed by APE-CACHE's distributed runtime. After introducing these two steps, we discuss our design considerations.

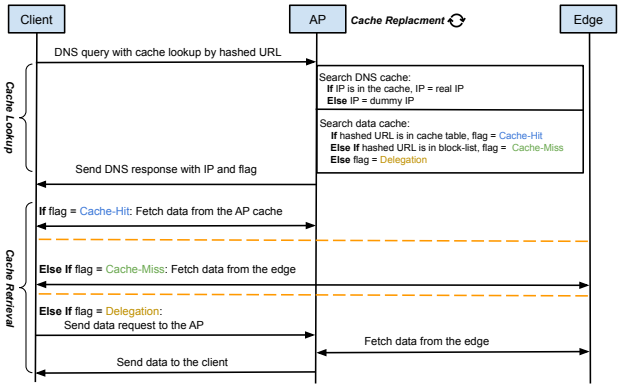


Fig. 7: Cache Lookup and Fetching Workflow

1) Cache Lookup by Piggybacking into DNS queries: As mentioned earlier, a client relies on DNS queries to identify the closest edge cache server. Our AP-based cache lookup method integrates with these DNS requests, to minimize extra overhead in scenarios in which the AP-based cache misses, and the client needs to revert to the edge-based cache. Overall, we rely on DNS infrastructure and its built-in extensibility support to customize queries, thus enabling APE-CACHE's cache lookup.

DNS-Cache Message Format: RFC1035 [13] defines a DNS message as consisting of multiple sections (i.e., Header, Question, Answer, Authority, and *Additional*) and one more multiple "Resource Record" (RR) items. The *Additional* section provides additional information related to DNS queries. For example, EDNS [14] creates a new RR type called *OPT* and uses *Additional* to transfer its corresponding information. We also utilize the "`Additional`" field in the DNS message, to transfer cache lookup requests and responses, and refer to the modified message as **DNS-Cache** queries.

```
<NAME>     ::= HOSTNAME
<TYPE>     ::= 300(DNS-Cache)
<CLASS>    ::= REQUEST/RESPONSE
<TTL>      ::= NULL
<RDLENGTH> ::= The Length of RDATA
<RDATA>    ::= [<HASH(URL),FLAG>]
```

Fig. 8: RR Format Definition for DNS-Cache Queries

We define a new RR type, as shown in Fig. 8. The field `<NAME>` stores the hostname. For the field `<TYPE>`, we assign an unsigned integer of 300 to indicate a "DNS-Cache" query. The field `<CLASS>` can be either "REQUEST" or "RESPONSE". The field `RDATA` stores a list of two-tuples $\langle HASH(URL), FLAG \rangle$, which will be introduced next. The field `RDLENGTH` is the bytes size of `RDATA`, used to parse `RDATA` on the client side.

DNS-Cache Request: In APE-CACHE, the client first sends a modified DNS request (DNS-Cache request) to the APE-CACHE AP. The DNS request sets `CLASS` as *REQUEST* in the "`Additional`" field of the DNS message, sets `NAME` as the domain name of the request, and puts the hashed URL of the request in the `RDATA` field. We hash the URL to maintain confidentiality, as DNS messages are unencrypted.

DNS-Cache Response: Upon receiving a DNS-Cache request, the AP searches for all URLs with the same domain name in the request. For each URL, the AP returns a `flag` to reflect its cache status, which can be divided into three categories: 1) `flag = Cache-Hit`: the data is stored in the cache and available for direct fetching; 2) `flag = Delegation`: the data is not stored in cache but the AP can delegate the client's request. It happens when the data has expired, or the hashed URL hasn't been seen by the AP before; 3) `flag = Cache-Miss`: the data is not available from the AP. It happens when the AP has delegated the request before but decided not to cache it anymore by adding it to a block list. If the data size exceeds a threshold (set at 500kb in our implementation), it will be added to the block list.

2) Fetching: After the cache lookup, the client retrieves the required data either from the WiFi AP or a remote server based on the DNS-Cache Response. It first hashes the URL of the required object and finds its `flag` from the received `RDATA`

field. If `flag = Cache-Hit`, the client sends a follow-up HTTP or HTTPS request to fetch the cached data from the AP; if `flag = Cache-Miss`, the client will continue with the original HTTP request to fetch the data the remote server using the IP address it receives from the DNS-Cache response; if `flag = Delegation`, the client sends the raw URL of the request, along with its TTL and priority level, to the AP. The AP then requests the data from the remote server, caches it, and sends it back to the client.

3) Design Considerations: Modifying the DNS module on APs may impact the original DNS query workflow. To resolve potential issues, we made certain accommodations.

- *Batching Multiple Cache Requests for the Same Domain*: apps often send several subsequent requests with identical domain names but different URLs. The client typically issues a single DNS request, as the DNS result is cached until expiration. In such a scenario, the subsequent cache requests cannot be issued in the absence of corresponding DNS requests. To handle this problem, we modified the AP runtime to respond with the cache status for all URLs under the same domain name.
- *Handling DNS resolution latency*: The inherent latency of DNS lookups can slow down the piggybacked cache lookup. Particularly, if the AP does not recognize the domain name, resolving its IP address from the upstream DNS server could introduce significant delays. To minimize this latency, if all URLs associated with a domain name are available on the AP, the AP circumvents waiting for DNS resolution by returning a dummy IP address in the DNS response and setting its TTL to 0.
- *Avoiding Sending Additional Requests*: To further reduce latency, many cache systems periodically refresh their cached data. However, in our scenario, the sheer volume of APs significantly outnumbers edge-based cache servers. Consequently, regularly updating the cache in such a manner would excessively strain the remote server. Therefore, APE-CACHE only sends a request to the remote server when triggered by the client, which prevents the escalation of the server workload.

C. Priority-Aware Cache Management

To delegate a user request, our AP runtime fetches the object from an edge or cloud server and subsequently caches it. In instances where the AP's cache is full and a new object arrives, our Priority-Aware Cache Management (PACM) algorithm dynamically determines which cached objects to evict, thereby creating sufficient space for the new object. **Two principal design objectives** guide PACM: 1) ensure that the retained objects in the cache deliver maximum benefits to nearby clients, and 2) achieve equitable distribution of the limited cache space across all apps.

System Modelling: Let C denote the cache capacity on an AP, and S denote the size of the new object. We assume $\mathcal{A} = \{a = 1, 2, ...A\}$ apps. We use $\mathcal{D} = \{d = 1, 2, ...D\}$ to denote all the objects currently cached on the AP. For each object d, A_d denotes the app it belongs to, p_d denotes its *priority* level, e_d denotes its *remaining valid time*, and l_d denotes the time a client saves by retrieving the object from AP, instead of from a remote server. All these parameters are either directly given by the developers or can be calculated by the AP: p_d is a positive integer provided by developers, e_d is calculated by the difference between the current time and the expiration time of d, and l_d is approximated by the latency of retrieving the object from the edge or cloud server.

We use $r_a(t_2 - t_1)$ to denote the number of requests for app a received by the AP from all clients between time t_2 and t_1. We calculate the **request frequency** $R(a)$ for app a as: $R(a) = (1 - \alpha) \times R'(a) + \alpha \times r_a(\Delta t)$, where $R'(a)$ is the $R(a)$ value in the previously round of calculation, Δt is the time since the previous calculation, and α (settled as 0.7 in our implementation) is a parameter that controls how the most recent measurements and past measurements contribute to the request frequency.

Problem Formulation: We define the **utility** of a data object d as $U_d = R(A_d) \times e_d \times l_d \times p_d$, where $R(A_d) \times e_d$ calculates the amount of future requests before the data expires, $l_d \times e_d$ calculates the impact of latency reduction for each request. We define the storage efficiency of an app a as $C_a = \frac{\sum_{\forall A_d = a} s_d}{R(a)}$, which means the storage space the app uses divided by its request frequency. To provide sufficient storage space for each app, we use Gini coefficient [15], a widely adopted approach for measuring fairness. We calculate the *fairness* of storage efficiency as:

$$F(\mathcal{A}) = \frac{\sum_{x=1}^{A} \sum_{y=1}^{A} |C_x - C_y|}{2A \times \sum_{x=1}^{A} C_x} \tag{1}$$

The output of PACM is $\mathcal{O} = \{O_d \in \{0, 1\}, \forall d \in \mathcal{D}\}$, where 0 means d will be evicted and 1 means otherwise. Considering both to deliver maximum benefits to nearby clients and to achieve equitable distribution of the limited cache space across all apps, we model the object selection problem as a two-dimensional Knapsack problem as given below:

$$\begin{aligned} &\mathcal{O} = \arg\max \sum_{d=1}^{D} O_d \times U_d \\ &s.t. : \sum_{d=1}^{D} O_d \times s_d \leq (C - S) \\ &s.t. : F(\mathcal{A}) \leq \theta \end{aligned} \tag{2}$$

, where $\sum_{d=1}^{D} O_d \times s_d \leq (C - S)$ means the size of all objects to be kept in the cache should not exceed the total cache capacity minus the size of the new object, and θ (settled as 0.4 in our implementation) is the threshold for the fairness index. Utilizing dynamic programming, PACM identifies the optimal set $\mathcal{O}$ that offers the best utility within the given constraints and accordingly evicts data objects from the cache.

D. Reference Implementation

For the clients, we extended `OkHttp (version 3.14.9)` [16], an open-source Android HTTP client. To

interface with DNS, we modified `c-ares (version 1.18.0)` [17] and integrated it into the `OkHttp`. For the AP, we extended `Dnsmasq (version 2.85)` [18], the built-in DNS library of OpneWRT [19], an open-source, Linux-based operating system widely used by off-the-shelf WiFi APs. Our reference implementation contains about $\sim$ 500 lines of C code and $\sim$ 400 lines of Java code for cache modules on Android, and $\sim$ 1200 lines of C code for modifying the DNS on AP.

V. Evaluation

Our evaluation questions alongside our summarized findings are as follows:

EQ1: What is APE-Cache's latency impact on retrieving a single cacheable object?

Finding: APE-Cache reduces the latency of retrieving a single cacheable object between 52% and 75%, as compared with two baseline approaches (Sec. V-B).

EQ2: What is APE-Cache's impact on app-level latency?

Finding: APE-Cache reduces app-level latency for real apps between 44% and 76%, as compared with two baseline approaches. (Sec. V-D).

EQ3: What are APE-Cache's usage overhead, in terms of its resource consumption on WiFi APs and programming effort?

Finding: APE-Cache only incurs at most an additional 6% of CPU utilization and uses only 13MB of memory on WiFi APs. APE-Cache requires no modification to the app logic.

A. Experimental Setup

We evaluated APE-Cache in realistic settings. To assess the performance impact of APE-Cache as compared to baseline approaches, we generated a suite of 30 mobile apps. This suite includes two apps modified from existing real-world apps and twenty-eight built from scratch, different in characteristics such as logic complexity, object size, and request latency. We then executed these subject apps on phones and simulators.

Baseline Methods: We compared APE-Cache with the following baseline solutions:

- *Edge Cache:* Caching data on edge servers in the vicinity of clients. The cached data objects are accessed by the edge server's domain name.
- *Wi-Cache:* Wi-Cache [20], an approach closely aligned with ours, was initially developed for caching large files, such as video chunks, across APs in a distributed manner. In this system, cache requests are initially routed to a centralized cache controller that identifies the nearest AP containing the required data to redirect the request accordingly. In our implementation, we retained and adapted the workflow of Wi-Cache for retrieving smaller cacheable objects instead of distributing large files. Besides, we kept its cache management algorithm, evicting the least-recently-used (LRU) data objects in the cache.
- APE-Cache-*LRU:* This approach follows the same workflow as APE-Cache but differs from its cache management algorithm, utilizing the LRU. We used this approach to evaluate the performance improvements offered by PACM.

APE-Cache, APE-Cache-LRU, and Wi-Cache are all specifically designed for deployment on APs. In our evaluation, if the required object is not present on the AP, all three systems were configured to redirect the request to an edge server for retrieval. We operated under the assumption that the edge server's cache capacity was ample enough to store all cacheable objects, thereby eliminating the need for cache replacement.

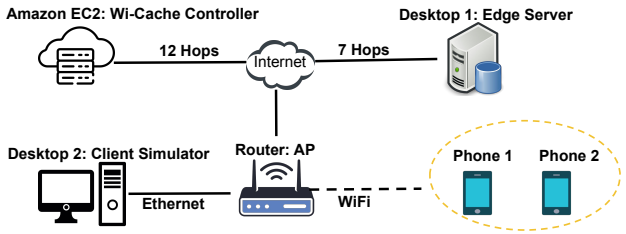


Fig. 9: Testbed Used for Evaluation

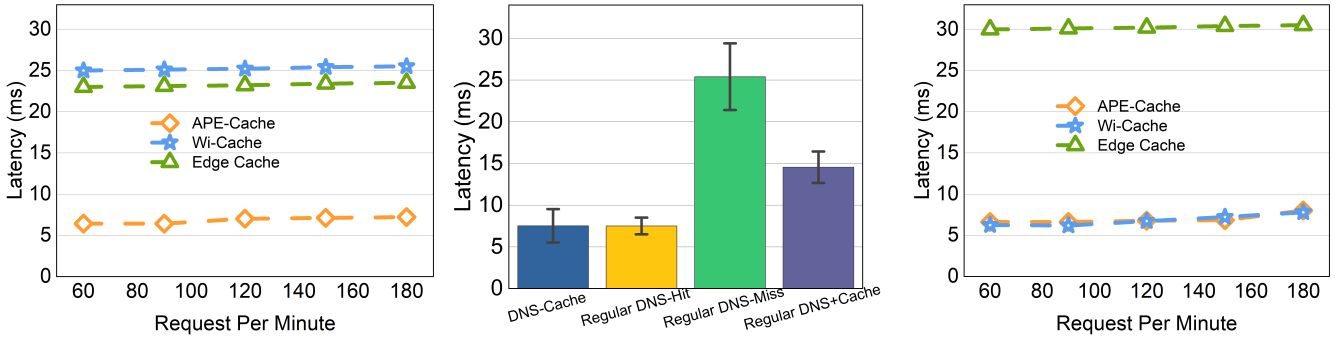
Deployment: We established a testbed as depicted in Fig. 9. Our setup included a commercially available router (GL-MT1300, equipped with an MT7621A CPU at 880MHz and 256MB Memory) functioning as the WiFi AP running APE-Cache. Two Android mobile phones (Samsung Galaxy A53, with Octa-core CPUs and 6GB RAM) and a desktop were directly connected to the AP. Each Android phone ran a real-world app, while the desktop operated an Android emulator to run the remaining twenty-eight apps. We deployed another desktop (Intel i7-4790, 16GB Memory) 7 hops away from the AP as the edge cache server. Furthermore, we rented an Amazon EC2 instance in Ohio as the Wi-Cache controller, which was 12 hops away from our AP.

app Name	Functionality	Critical Path
`MovieTrailer`	Given a movie name, display its detailed information	MovieName → movieID → rating, plot, cast, thumbnail
`VirtualHome`	Given a product gallery, display its furniture gallery	selectProduct → ARObjectsID → ARObjects

Fig. 10: Functionality and Critical Path of Real-world apps

Apps and Execution: We chose two latency-sensitive real-world apps in our evaluation, i.e., `MovieTrailer` [12] from our motivating example and `VirtualHome` [21]. Both apps are open-sourced on GitHub and available on Google Play. Fig.10 describes their main functionalities and critical paths. When executing `MovieTrailer`, we randomly chose one movie name from the top 10 movie names on IMDB website [22] as the user input, to trigger the follow-up workflow. Similarly, we randomly chose a product category to trigger the follow-up workflow for `VirtualHome`. Table III lists the cacheable objects and their priorities for these apps.

To expand our evaluation, we developed a dummy app generator and synthesized 28 apps with specific characteristics based on given input parameters. For each app, we generated cacheable objects with randomly assigned attributes, including



(a) Cache Lookup Latency (b) Lookup Latency Overhead (c) Cache Retrieval Latency

Fig. 11: Object-level Caching Latency

	High Priority	Low Priority
MovieTrailer	movieID, thumbnail	rating, plot, cast
VirtualHome	ARObjects	ARObjectsID

TABLE III: Priority Values in Real-world apps

size, TTL, and retrieval latency. These objects were hosted on our edge server, with an added delay (retrieval latency) to simulate the latency experienced when retrieving them from various servers. The retrieval latency was set to range between 20ms and 50ms, TTL varied from 10 minutes to 60 minutes, and object sizes spanned from 1kb to 100kb. The priority for each object was assigned as 1 or 2 based on the critical path of the app. We adopted the Zipf distribution [23] to calculate the time interval between executing an app. As a result, different apps exhibited dissimilar usage frequencies, and the average frequency for all apps was set to 3 times per minute.

B. Object-Level Caching Latency

Object-level caching latency refers to the end-to-end time for a client to retrieve a cacheable object from the cache at an edge server or WiFi AP. Since APE-CACHE and the other two baselines follow the same two-stage object delivery workflow (i.e., cache lookup and cache retrieval), we measured the latency of each stage separately and added them up as the overall latency of retrieving a cacheable object.

- *Cache Lookup Latency:* Cache lookup latency refers to the duration required for clients to determine the cache status. More precisely, it encompasses the time taken to resolve the IP address of the edge caching server, for Wi-Cache to identify the appropriate AP to serve the client, and for APE-CACHE to complete a DNS-Cache query.
- *Cache Retrieval Latency:* Cache retrieval latency is the period from when a request for an object is sent to the cache during a hit (e.g., `flag=Cache-Hit` for APE-CACHE) to when the object's response is received by the client. The duration is measured from initiating a TCP session between the client and the cache to the point that the client's socket starts to read the object's content.

Due to AP's limited processing power, we varied the usage frequency for all apps to assess how request workload impacts the object-level caching latency. For each usage frequency, we executed the apps for an hour and recorded the object retrieval latency for different caching solutions, with the AP cache size set to 5MB.

Cache Lookup Latency: We first compared APE-CACHE's cache lookup latency with that of the baseline methods. As Fig. 11a shows: 1) for all three caching methods, the latency rises in response to increased request frequency, with the rise being rather insignificant (e.g., 0.5ms in APE-CACHE); 2) APE-CACHE exhibited significantly lower latency, averaging around 7.5ms, while both Wi-Cache and Edge Cache exceeded 22ms. While Wi-Cache contacts a remote controller to find the appropriate AP, Edge Cache communicates with a remote DNS resolver to locate the nearest cache server.

The millisecond-level cache lookup latency is enabled not only by directly communicating with AP, but also by APE-CACHE's unique DNS-Cache design, whose performance characteristics appear in Fig. 11b. Comparing the latencies of DNS-Cache queries with regular DNS queries (hit), where the requested domain name is cached on the AP, revealed that the DNS-Cache query introduced a mere additional 0.02 ms of latency. A regular DNS query's latency increases steeply due to recursive DNS queries to locate the missing DNS record at AP. In contrast, our DNS-Cache query circumvents this inefficiency by bypassing DNS resolution when the requested objects are cached on the AP. Further, we compared the latency of piggybacking DNS-Cache queries with sending two standalone queries, i.e., a cache query after a regular DNS query. We observed that separating two queries incurred an additional latency overhead of 7.02 ms, as compared with the integrated design of our DNS-Cache querying. Therefore, the innovative DNS-piggybacking and the short-circuited return of cache status contribute to APE-CACHE's millisecond-level cache lookup latency.

Cache Retrieval Latency: As shown by Fig. 11c, APE-CACHE and Wi-Cache had a similar latency of 7 ms, much lower than Edge Cache's 30 ms latency. The advantage of APE-CACHE and Wi-Cache lies in their proximity to app users, as they cache objects directly on the AP. This proximity reduces latency for establishing TCP connections

and transmitting data objects, compared to retrieving data from a more distant edge server. In contrast, the increase in request frequency impacted the retrieval latency more than the lookup latency, as handling TCP-based retrieval requests is generally more resource-intensive than handling UDP-based lookup requests.

Summary: By summing the latencies of cache lookup and retrieval, the overall latencies for retrieving a single object from APE-CACHE, Wi-Cache, and Edge Cache were 14.24 ms, 29.50 ms, and 55.93 ms, respectively. APE-CACHE reduced the latency by 51.72% and 74.54% compared with the two baseline solutions. Request frequency seems to have an unnoticeable impact on APE-CACHE's performance.

C. Performance of PACM

We compared the performance of two cache management algorithms, PACM used by APE-CACHE and LRU used by Wi-Cache and APE-CACHE-LRU. We varied factors such as object size, app usage frequency, and the number of apps, to simulate the impact of 1) larger data objects, 2) fewer users connected to the AP, and 3) a more diverse range of apps. We used the cache hit ratio as the performance indicator and measured such ratio separately for requests retrieving high-priority objects and all data objects. As object-level latency for retrieving data from AP is significantly lower, more cache hits indicate a higher latency reduction, especially for high-priority objects. In this experiment, we set the default parameters as: $1 \sim 100kb$ for object size, 3 for app usage frequency, and 30 for the number of apps.

Data Object Size	PACM-Avg	PACM-High Priority	LRU
1∼ 100 kb	0.632	0.832	0.631
1∼ 200 kb	0.514	0.754	0.528
1∼ 300 kb	0.426	0.616	0.430
1∼ 400 kb	0.320	0.457	0.316
1∼ 500 kb	0.226	0.304	0.220

TABLE IV: Cache Hit Ratio vs. Data Object Size

Impact of Object Size: Table IV shows the correlation between the cache hit ratio and data object size for these two methods. We observed that as the object size expands, the hit ratio of both approaches decreases. For high-priority objects, PACM consistently maintained a higher hit ratio. As the object size increases, the limited cache size can fit fewer objects, so PACM needs to evict low-priority objects.

Avg. Frequency	PACM-Avg	PACM-High Priority	LRU
1	0.507	0.743	0.512
1.5	0.563	0.766	0.566
2	0.626	0.774	0.625
2.5	0.627	0.810	0.628
3	0.632	0.832	0.631

TABLE V: Cache Hit Ratio vs. Avg. app Usage Frequency

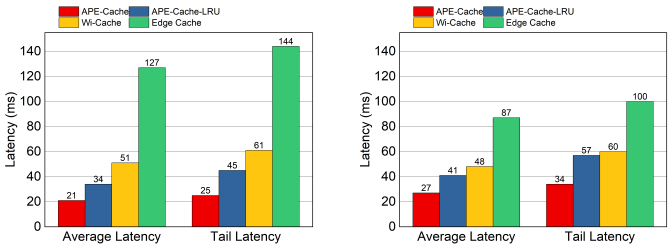
Impact of App Usage Frequency: When fewer users of the same app connect to the AP, app usage frequency decreases. This decrease can cause cached objects to expire before they are requested again, resulting in cache misses. As indicated in Table V, lowering the frequency decreased the cache hit ratio, albeit insignificantly. Even so, PACM still ensured a higher hit ratio for high-priority objects compared to LRU.

App Quantity	PACM-Avg	PACM-High Priority	LRU
5	0.965	0.965	0.965
10	0.966	0.966	0.966
15	0.967	0.945	0.967
20	0.763	0.889	0.765
25	0.691	0.841	0.668
30	0.632	0.832	0.631

TABLE VI: Cache Hit Ratio vs. App Quantity

Impact of App Quantity: As the range of used apps increases (as indicated by the higher quantity of apps in Table VI), more objects need caching. Due to the limited cache memory, some objects got evicted, thus causing more cache misses. Nevertheless, PACM consistently maintained a higher hit ratio for the high-priority objects.

D. App-level Performance Improvements



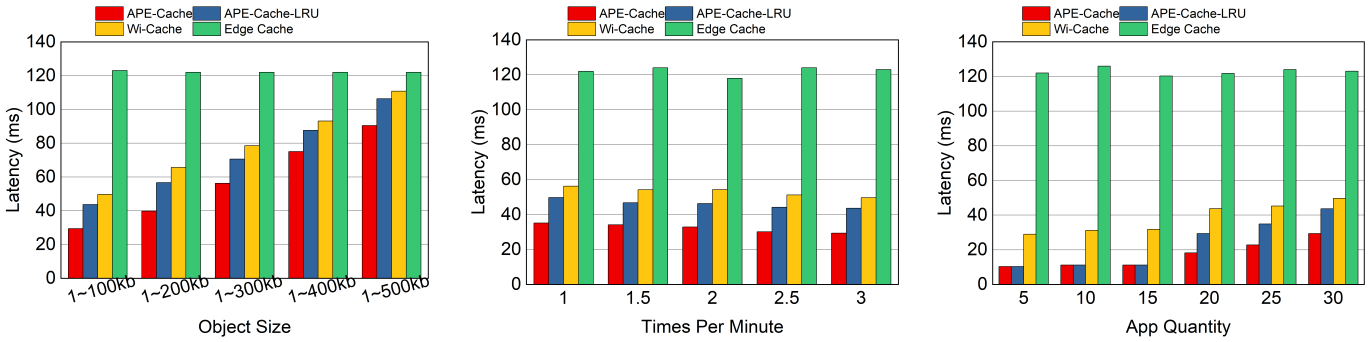
(a) MovieTrailer (b) VirtualHome

Fig. 12: Real-world apps' Latency Performance

Reducing app-level latency is instrumental in improving the user experience. Fig. 12 shows the average latency and tail latency (i.e., 95^{th} percentile latency) of the two real-world apps. We observed that APE-CACHE outperformed all baseline methods, especially Edge Cache, by reducing 78% of average latency and 76% of tail latency for both apps. APE-CACHE's reduction of app-level latency is due to: 1) decreasing the latency of retrieving objects from AP; 2) increasing the cache hit ratio for high-priority objects; 3) incurring negligible latency overhead even in the presence of cache misses on AP.

As item 2) can be affected by data object size, app usage frequency, and app quantity, we further compared the app-level latency of APE-CACHE with the baselines methods, varying these parameters in the same way as in Sec. V-C. Fig. 13 shows the average app-level latency of all 30 apps under various settings, with APE-CACHE outperforming the baseline methods across the board. At the default parameters setting, the average app-level latency of APE-CACHE, APE-CACHE-LRU, Wi-Cache, and Edge Cache were 30ms, 42ms, 54ms, and 122ms, respectively. APE-CACHE reduced the latency by 29%, 44% and 76% compared with the baselines.

Besides, the results also confirmed our assumptions in Sec. V-C: 1) A higher cache hit ratio reduces app-level latency. Decreasing the object size or the number of apps led to a higher cache hit ratio, thus reducing the latency; 2) A higher cache hit ratio for high-priority objects can further reduce the app-level latency. Indeed, APE-CACHE consistently exhibited



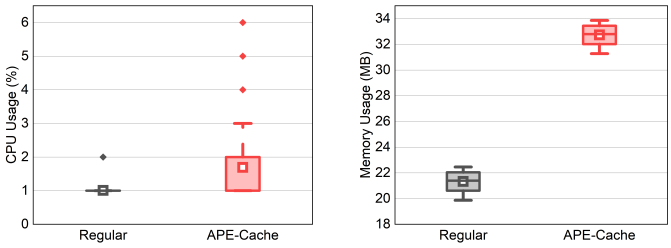
(a) Latency vs. Data Object Size (b) Latency vs. App Usage Frequency (c) Latency vs. App Quantity

Fig. 13: Average App-Level Latency of All Apps Under Various Experiment Settings

lower latency than APE-CACHE-LRU, as APE-CACHE-LRU is essentially APE-CACHE but without differentiating the priority of objects.

E. Overheads of APE-CACHE

To enable caching on APs, APE-CACHE introduces additional components whose impact on AP resource consumption. We evaluated this impact to ensure that AP's core functionalities (e.g., IP routing and packet forwarding) are not adversely affected. To this end, we used 30 pairs of apps, each consisting of a APE-CACHE-enabled version and a regular version. The regular versions were connected to the AP without being modified and directly retrieved objects from the edge server. We allocated 5MB cache memory on AP for APE-CACHE. We ran APE-CACHE-enabled apps and regular apps at a frequency of 3 times per minute for one hour and recorded their CPU and memory usage during this period.



(a) CPU Usage (b) Memory Usage

Fig. 14: CPU/Memory Usages on WiFi AP

As shown in Fig. 14, APE-CACHE exhibited an increase in CPU usage (up to 6%) and memory usage (up to 13MB). Three factors explain these increases: 1) object caching functions effectively with only 5MB of memory; 2) processing DNS-Cache queries and HTTP requests required additional CPU and memory resources; 3) executing the PACM algorithm also consumed resources. However, even with these additional processes, APE-CACHE's overhead remained modest, as compared to the resources of modern APs, confirmed by our study in Sec. II-C.

F. Programming Effort

To highlight the usability of our annotation-based programming model, we developed an alternative model that relies on API calls to access AP-based caching. In particular, the alternative uses our modified `c-ares` library to issue DNS-Cache queries and defines a specific API for asynchronously issuing HTTP requests for cacheable objects: `String invokeHttpRequestAsync(String url, int priority, int TTL)`. In this model, developers have to modify the app's logic by replacing original HTTP requests with this API.

App	Approach	Impacted LoCs	Extra Binary Size	Re-write Logic
MovieTrailer	APE-CACHE	5	32kb	No
	API-based	30	32kb	Yes
VirtualHome	APE-CACHE	2	32kb	No
	API-based	14	32kb	Yes

TABLE VII: Programming Efforts Comparison

We compared the programming efforts of these two approaches both quantitatively and qualitatively. Quantitatively, we assessed the impact on the number of lines of codes (LoCs) and the additional app binary size relative to the original binary size. Qualitatively, we examined the ease of use for developers (e.g., re-writing code and debugging work). As indicated by Table VII, APE-CACHE required only a few code annotations, as opposed to the API-based approach, which necessitated modifications to the app's logic. Specifically, all HTTP requests associated with cacheable objects needed rewriting, thus increasing the likelihood of introducing insidious errors that would be non-trivial to fix. Regarding the additional size, both approaches resulted in a comparable and reasonable binary increase of about 32kb.

Although annotations are specific to Java, our programming model is designed to be compatible with any language that supports declarative metadata [24]. This metadata assigns specialized meaning to programming constructs, such as class fields, enabling external tools to seamlessly enhance code functionality. Enterprise software development has embraced this programming model through metadata-driven frameworks and libraries [25]. Some state-of-the-art approaches even advocate for techniques that facilitate the reuse of metadata information across programming languages [26], [27].

VI. Related Work

APE-Cache is related to edge caching systems, priority-aware caching, and mobile app acceleration. Next we discuss the related approaches in these domains in turn.

Edge Caching Systems Caching data at the network edge significantly reduces latency. One commonly used edge caching system is content delivery networks (CDN) [28] that cache the data in cache servers geographically co-located with the end users, thereby reducing latency. To further accelerate data delivery, many works cache data at cellular network base stations [29], [30] and WiFi APs [31], [32], which are extremely close to users, with few practical implementations. One such implementation is Wi-Cache [20], which caches video chunks on WiFi APs to accelerate the process of retrieving large video files for mobile clients.

However, these caching systems suffer from a high latency overhead in cache lookup. CDN and base station caching require contacting a remote DNS server to direct requests to the nearest cache server or base station, while Wi-Cache needs to consult a controller to identify the appropriate AP. APE-Cache piggybacks the cache lookup request into DNS queries to minimize latency. Although edge computing approaches [33], [34] have relied on WiFi AP for coordinating and executing computing tasks, our approach marks the first to intricately integrate supplementary functionality with the AP's inherent capabilities. This pioneering integration opens up a novel direction in edge computing research.

Priority-Aware Caching Several studies customized caching for CDN that take into account both data usage frequency and priority [35]–[37]. However, these theoretical solutions primarily focus on web pages, whose complex dependencies necessitate varied cache priorities for different web objects.

As mobile apps have become part and parcel of our lives, users tend to regularly utilize mobile apps alongside web pages. More importantly, network-intensive apps exhibit complex dependencies between their data requests [38]. APE-Cache's novelty lies in its priority-aware caching that is specifically tailored for mobile apps. It also offers a declarative programming model to conveniently define object priority. Finally, its cache management scheme strategically places cacheable objects based on their priorities, thereby optimizing the utilization of limited AP cache memory.

Mobile App Acceleration Multiple mobile systems accelerate their performance via prefetching. They differ in their strategies for deciding the specifics of prefetching——what data to fetch, when to fetch it, and from where. For example, APPx [38] utilizes static analysis techniques to identify dependencies between requests and then sends this dependency information to an edge proxy for prefetching the dependent data. Similarly, PALOMA [39] and Marauder [40] prefetch the data objects, but do so on the client side and store them on the device for future access. APE-Cache is orthogonal to these efforts and can be combined to further reduce app latency: by enabling apps with client-side prefetching to source data from a APE-Cache-enabled AP, or by sending the request dependency information to the APE-Cache-enabled AP to prefetch data, thereby reducing cache misses.

VII. Conclusion

This paper presented APE-Cache, an edge caching system that taps into pervasive networking infrastructure to cache data for mobile apps. To the best of our knowledge, APE-Cache is the first AP-based caching system that complements existing edge caching solutions, thus making it practical for deployment on off-the-shelf APs. Our evaluation demonstrated that APE-Cache effectively reduces app-level latency for real apps by as much as 76%, as compared to CDN, while imposing minimal resource and programming effort overheads. As new and exciting time-sensitive applications are starting to emerge, APE-Cache has the potential to become a powerful and cost-efficient system component for their construction.

VIII. Acknowledgement

This research is supported by NSF through the grants #2104337 and # 2232565.

References

[1] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, "Toward low-latency and ultra-reliable virtual reality," *IEEE Network*, vol. 32, no. 2, pp. 78–84, 2018.

[2] Z. Amjad, A. Sikora, B. Hilt, and J.-P. Lauffenburger, "Low latency v2x applications and network requirements: Performance evaluation," in *2018 IEEE IV*. IEEE, 2018, pp. 220–225.

[3] G. Kolovou, S. Oteafy, and P. Chatzimisios, "A remote surgery use case for the IEEE p1918. 1 tactile internet standard," in *ICC'21*, pp. 1–6.

[4] NVIDIA, "NVIDIA CloudXR suite," 2022. [Online]. Available: https://www.nvidia.com/en-us/design-visualization/solutions/cloud-xr/

[5] J. Han, J. Davids, H. Ashrafian, A. Darzi, D. S. Elson, and M. Sodergren, "A systematic review of robotic surgery: From supervised paradigms to fully autonomous robotic approaches," *The International Journal of Medical Robotics and Computer Assisted Surgery*, vol. 18, no. 2, 2022.

[6] AWS, "Aws autonomous mobility," 2022. [Online]. Available: https://aws.amazon.com/automotive/autonomous-mobility/

[7] H. Zhang, S. Huang, M. Xu, D. Guo, X. Wang, V. C. Leung, and W. Wang, "How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild," in *IWQoS'23*, 2023, pp. 1–10.

[8] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: a first look at public edge platforms," in *ACM IMC'21*, 2021, pp. 37–53.

[9] K.-J. Hsu, J. Choncholas, K. Bhardwaj, and A. Gavrilovska, "DNS does not suffice for MEC-CDN," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 212–218.

[10] L. Huang, S. Cheng, Y. Guan, X. Zhang, and Z. Guo, "Consistent user-traffic allocation and load balancing in mobile edge caching," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2020, pp. 592–597.

[11] Tcpreplay Suite, "Sample captures of network traffic," https://tcpreplay.appneta.com/wiki/captures.html.

[12] MovieTrailer, https://github.com/marwa-eltayeb/MovieTrailer, accessed: 2023-03-15.

[13] P. Mockapetris, "Domain names - implementation and specification," 1987. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1035

[14] IETF, "Extension mechanisms for DNS (EDNS(0))," https://datatracker.ietf.org/doc/html/rfc6891, accessed: 2022-05-29.

[15] A. Druckman and T. Jackson, "Measuring resource inequalities: The concepts and methodology for an area-based gini coefficient," *Ecological economics*, vol. 65, no. 2, pp. 242–252, 2008.

[16] okhttp, https://square.github.io/okhttp/.

[17] c-ares, "A C library for asynchronous DNS requests," https://c-ares.org.

[18] S. Kelley, "Dnsmasq," https://thekelleys.org.uk/dnsmasq/doc.html, 2021.

[19] Wikipedia, "Openwrt," https://en.wikipedia.org/wiki/OpenWrt.

[20] L. Chhangte, N. Karamchandani, D. Manjunath, and E. Viterbo, "Towards a distributed caching service at the wifi edge using wi-cache," *IEEE TNSM*, vol. 18, no. 4, pp. 4489–4502, 2021.

[21] VirtualHome, https://github.com/rkswetha/VirtualHome.

[22] IMDB, "IMDB top 100 (sorted by user rating descending)." https://www.imdb.com/search/title/?sort=user_rating,desc&groups=top_100.

[23] M. Kihl, R. Larsson, N. Unnervik, J. Haberkamm, A. Arvidsson, and A. Aurelius, "Analysis of facebook content demand patterns," in *2014 International Conference on Smart Communications in Network Technologies (SaCoNeT)*. IEEE, 2014, pp. 1–6.

[24] M. Song and E. Tilevich, "Metadata invariants: Checking and inferring metadata coding conventions," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 694–704.

[25] ——, "Enhancing source-level programming tools with an awareness of transparent program transformations," *SIGPLAN Not.*, vol. 44, no. 10, p. 301–320, oct 2009. [Online]. Available: https://doi.org/10.1145/1639949.1640112

[26] E. Tilevich and M. Song, "Reusable enterprise metadata with pattern-based structural expressions," in *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, ser. AOSD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 25–36. [Online]. Available: https://doi.org/10.1145/1739230.1739234

[27] M. Song and E. Tilevich, "Reusing metadata across components, applications, and languages," *Science of Computer Programming*, vol. 98, pp. 617–644, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642314003682

[28] A.-M. K. Pathan, R. Buyya *et al.*, "A taxonomy and survey of content delivery networks," *University of Melbourne, Technical Report*, vol. 4, no. 2007, p. 70, 2007.

[29] N. Golzerai, K. Shanmugam, A. Dimakis, A. Molisch, and G. Caire, "Femtocaching: wireless video content delivery through distributed caching helpers," in *Proc. IEEE INFOCOM*, 2012.

[30] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5g wireless networks," *IEEE Communications Magazine*, vol. 52, no. 8, pp. 82–89, 2014.

[31] F. Lyu, J. Ren, N. Cheng, P. Yang, M. Li, Y. Zhang, and X. S. Shen, "Lead: Large-scale edge cache deployment based on spatio-temporal wifi traffic statistics," *IEEE TMC*, vol. 20, no. 8, pp. 2607–2623, 2020.

[32] K. Poularakis, G. Iosifidis, I. Pefkianakis, L. Tassiulas, and M. May, "Mobile data offloading through caching in residential 802.11 wireless networks," *TNSM*, vol. 13, no. 1, pp. 71–84, 2016.

[33] Z. Song and E. Tilevich, "PMDC: Programmable mobile device clouds for convenient and efficient service provisioning," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 202–209.

[34] ——, "A programming model for reliable and efficient edge-based execution under resource variability," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 64–71.

[35] Z. Wang, W. Zhu, M. Chen, L. Sun, and S. Yang, "CPCDN: Content delivery powered by context and user intelligence," *IEEE Transactions on Multimedia*, vol. 17, no. 1, pp. 92–103, 2014.

[36] S. Puzhavakath Narayanan, Y. S. Nam, A. Sivakumar, B. Chandrasekaran, B. Maggs, and S. Rao, "Reducing latency through page-aware management of web objects by content delivery networks," *ACM SIGMETRICS*, vol. 44, no. 1, pp. 89–100, 2016.

[37] H. Hu, Y. Li, and Y. Wen, "Toward rendering-latency reduction for composable web services via priority-based object caching," *IEEE Transactions on Multimedia*, vol. 20, no. 7, pp. 1864–1875, 2017.

[38] B. Choi, J. Kim, D. Cho, S. Kim, and D. Han, "Appx: an automated app acceleration framework for low latency mobile app," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, 2018, pp. 27–40.

[39] Y. Zhao, M. S. Laser, Y. Lyu, and N. Medvidovic, "Leveraging program analysis to reduce user-perceived latency in mobile applications," in *ICSE'18*, 2018, pp. 176–186.

[40] M. Ramanujam, H. V. Madhyastha, and R. Netravali, "Marauder: synergized caching and prefetching for low-risk mobile app acceleration," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 350–362.