Hit and Bandwidth Optimal Caching for Wireless Data Access Networks

by

Mursalin Akon

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

For many data access applications, the availability of the most updated information is a fundamental and rigid requirement. In spite of many technological improvements, in wireless networks, wireless channels (or bandwidth) are the most scarce resources and hence are expensive. Data access from remote sites heavily depends on these expensive resources. Due to affordable smart mobile devices and tremendous popularity of various Internet-based services, demand for data from these mobile devices are growing very fast. In many cases, it is becoming impossible for the wireless data service providers to satisfy the demand for data using the current network infrastructures. An efficient caching scheme at the client side can soothe the problem by reducing the amount of data transferred over the wireless channels. However, an update event makes the associated cached data objects obsolete and useless for the applications. Frequencies of data update, as well as data access play essential roles in cache access and replacement policies. Intuitively, frequently accessed and infrequently updated objects should be given higher preference while preserving in the cache. However, modeling this intuition is challenging, particularly in a network environment where updates are injected by both the server and the clients, distributed all over networks.

In this thesis, we strive to make three inter-related contributions. Firstly, we propose two enhanced cache access policies. The access policies ensure strong consistency of the cached data objects through proactive or reactive interactions with the data server. At the same time, these policies collect information about access and update frequencies of hosted objects to facilitate efficient deployment of the cache replacement policy. Secondly, we design a replacement policy which plays the decision maker role when there is a new object to accommodate in a fully occupied cache. The statistical information collected by the access policies enables the decision making process. This process is modeled around the idea of preserving frequently accessed but less frequently updated objects in the cache. Thirdly, we analytically show that a cache management scheme with the proposed replacement policy bundled with any of the cache access policies guarantees optimum amount of data transmission by increasing the number of effective hits in the cache system. Results from both analysis and our extensive simulations demonstrate that the proposed policies outperform the popular Least Frequently Used (LFU) policy in terms of both effective hits and bandwidth consumption. Moreover, our flexible system model makes the proposed policies equally applicable to applications for the existing 3G, as well as upcoming LTE, LTE Advanced and WiMAX wireless data access networks.

Acknowledgements

I would like to express my deep and sincere thanks to my supervisors Dr. Xuemin (Sherman) Shen and Dr. Ajit Singh, for being my supervisors. Their versatile knowledge have been a great value for me. Their guidance, supports, and encouragements at both professional and personal level, throughout my study at the University of Waterloo, made them more than my gurus.

My cordial thanks to my thesis committee members for their time and efforts. My sincere thanks to Dr. Ehab Elmallah for his detail and important comments, and editorial checking of the entire thesis draft. I owe my sincere gratitude to Dr. Liping Fu and Dr. Pin-Han Ho for their very useful comments. I warmly thanks to Dr. Sagar Naik for his helpfulness and for always keeping his door open for ideas and suggestions.

My warm thanks are due to my colleagues at the Broadband Communications Research (BBCR) group and Network Programming Lab (NPL). Their constant feedbacks, supports and encouragements are noteworthy. Besides, discussions and talks on diversified areas of communication and computer networks kept me up-to-date about cutting edge researches. I am very happy and satisfied for being part of the two best LABs at the University of Waterloo campus.

My sincere thanks to Mr. Islam for being a good friend and an excellent research collaborator.

I would like to thank Natural Sciences and Engineering Research Council (NSERC) of Canada for providing financial supports.

My loving thanks go to my parents, my wife and my kids. Without their their encouragements and understanding, it would be impossible for me to complete this thesis.

Finally, I am truly grateful to the great Almighty for everything good in my life. All praises are for him.

Some of the contents presented in this thesis have been published in [28]. A written permission from the publisher is not required to include those contents in this thesis.

Dedication

For my parents.

Table of Contents

List of Figures

Nomenclature

PAP	:	Proactive Access Policy
RAP	:	Reactive Access Policy
URP	:	Update-based Replacement Policy
LFU	:	Least Frequently Used
LRU	:	Least Recently Used
IR	:	Invalidation Report
LTE	:	Long Term Evolution
WiMAX	:	Worldwide Interoperability for Microwave Access
GF	:	Gain Factor
μ_i^j	:	Access rate to object i at client j
$\mu_i^{\mathbb{A}}$	:	$\sum_j \mu_i^j$
μ_i	:	$\mu_i^{\mathbb{A}}$ or $\sum_j \mu_i^j$
μ^j	:	$\sum_i \mu_i^j$
λ_i^j	:	Update rate to object i at client j
$\lambda_i^{\mathbb{S}}$	:	Update rate to object i at client j
$\lambda_i^{\mathbb{A}}$	:	$\sum_j \lambda_i^j$
λ_i	:	$\mu_i^{\mathbb{A}} + \mu_i^{\mathbb{S}}$

Chapter 1

Introduction

This chapter starts with an introduction to the problem of caching in computing environment in Section 1.1. We then describe the thrust behind further researches in caching in wireless environment in Section 1.2. The problems associated with this type of caching are given next. The chapter is concluded with an outline of the rest of the thesis and a list of expected contributions.

1.1 Caching in Computing Environment

The concept of caching is being used in computing environment to solve variety of problems. Traditionally, a cache is a component (either in software or in hardware) to transparently improve performance by storing recurring data objects such that future requests for these objects can be quickly served. A cached object might be a previously computed value or a copy of an object stored elsewhere. If a requested object is contained in the cache, the object is served by fetching from the cache. Such an event is called a *cache hit*. Otherwise, a *cache miss* takes place and the requested object values have to be recomputed or be retrieved from the original source. Typically, a cache miss results in a much expensive process (in terms of resources) than a cache hit.

A hardware cache in Central Processing Unit (CPU) is used to reduce the average time to access memory. The cache is a smaller but faster memory and is employed to store copy of data from the main memory. Modern CPUs are typically equipped with three caches: to speedup fetching executable instructions an instruction cache; to speedup data access a data cache; and to facilitate faster translation of virtual addresses to physical address a translation lookaside buffer (TLB) cache.

Unlike CPU caches, a variety of software components manage caches. One example

is disk cache or page cache. Disk cache is a buffer[1], kept in the main memory, for faster data access. Disk cache is typically managed by the operating system (i.e., kernel) and is transparent to applications. Disk seek and read time is magnitudes of longer than the read time from the main memory and this is the main motivation of employing disk cache, which contributes towards significant improvements in speed and responsiveness of a computing device. Many modern disks provide an extra level of cache hierarchy by integrating hardware caches within the disks.

Web browser is another example where software-based cache is used. The purpose of Web-caching is to increase responsiveness of the browser by serving previously stored Web pages. Web-caching may be implemented at the browser, at the proxy, or even at the server gateways. As will be discussed later on, in detail, caching in network environment often helps in reducing bandwidth requirements and lessen server load.

The concept of caching is heavy used in domain name system (DNS) [43] servers, network file systems, search engines and databases.

1.2 Motivation

Extraordinary advances in computing electronics and wireless communications promise flexibility to our daily lives. Traditional cellular devices was mostly used for voice communication, whereas, digital organizer devices was for storing details about personal contacts and schedules. In contrast, modern mobile devices, such as smart phones, personal digital assistants (PDAs) and other handheld computers, and deployed high bandwidth 3G (and expected 3.5G and 4G) cellular networks and wireless LANs create the platform for ubiquitous mobile computing. These technologies have made many necessary and entertaining mobile IP-based applications possible. Mobile IP telephony, mobile TV, video on demand, video conference, tele-medicine, instant messaging, mobile online banking, stock market tracking, online multi-player games are examples of such applications. Online social networking, such as, Facebook [16], Qzone [33], MySpace [30], Twitter [40]; video sharing sites, such as Youtube [50], Youku [49]; image and file hosts, such as Skydrive [44], Flickr [47], Picasa [19], Photobucket [32], Dropbox [15] have changed the way people communicate and store their multimedia and other documents. These applications and services consume so much bandwidth, burdening the communication infrastructures, that the service provides have no choice but look for alternate methodologies and user incentive mechanisms [2–4]. As prices of advanced mobile devices become more affordable and more users subscribe for wireless data access services, the problem of bandwidth is simply going to get worse.

[1]The terms "buffer" and "cache" are often used in close contexts. In this thesis, we consider a buffer as a temporary storage/memory location for further processing. A cache buffer is a temporary storage/memory blocks where cached contents are preserved.

In a mobile information retrieval system, databases and files are hosted at remote servers, conventionally connected on the wired networks. Each database or file server hosts a number of objects, made available to be accessed by the mobile users. Obviously, whenever a mobile user accesses a data object from the remote server, all the communications have to pass through the wireless network. In spite of the advancements in wireless technologies, wireless bandwidth is the most scarce and the most expensive resource in a wireless data network. A client has to be very economic about the bandwidth consumption and make the best effort towards higher utilization. Many data access applications adaptively adjust the quality of service depending on the network state. For example, a mobile Internet browser may retrieve images whose quality is adjusted to the available bandwidth. However, developing such network aware applications is not trivial [14]. Besides, these adaptive applications do not focus on reducing bandwidth consumption, rather try to maximize the consumption depending on the wireless channel and other conditions. Many applications can reduce bandwidth consumption by caching recurrently accessed objects locally. Note that, cache oriented solutions are not orthogonal to developing network aware applications, rather in most cases cache can be deployed irrespective of network awareness.

Caching contributes in three ways to improve the performance of the data access applications and the network systems. Firstly, the average access latency is reduced as many data objects are delivered from the local cache, instead of fetching them from the remote server. Secondly, without cache, at each access, an object has to be fetched from the server and the object has to be passed through the network from the server to the client. Thus caching reduces the network load. Reduced network load decreases the cost of data access. Thirdly, as the server gets fewer request from a client, the server becomes more scalable without additional computing and network resources. An interesting side effect of employing cache is that by cutting down the number of communication transmissions, caching not only salvages on expensive wireless access but also saves power and prolongs battery life[2].

1.3 Problems

The goal of deploying a cache may vary from one application to another. While deploying a cache, an application needs to decide about policies to handle different aspects. A set of selected policies to manage a cache for a particular application is termed as a cache scheme. In this research, we are interested in two policies of a cache scheme – a cache *access*; and a *replacement* policy. A cache access policy describes two important jobs of a

[2]It is worth noting that transmission over wireless data network consumes a good amount of power [31]. However, reducing power consumption is not the main focus, but is an attractive by product of this research.

cache mechanism – firstly, how a client and a server utilize the cache and; secondly, how consistency between the original data items at the server and copies at the client caches are maintained. In distributed environment, the second task is often very complicated and termed as cache *consistency* or *invalidation* policy, if investigated from data consistency or data invalidation perspective, respectively.

For many data access applications, existence of a more recent update renders all older copies of the data object invalid. These applications must have access to the most recent updated data. This kind of consistency is called the *latest value* consistency [9, 42], and a cache, satisfying the latest value consistency, is said to be *strongly consistent* [9, 25]. With the latest value consistency requirement, when a data object is updated, all the cached copies become obsolete and can not be used to server application requests. In this case, a client has to retrieve the data item from the server. Several strongly consistent cache consistency policies for wireless data access have been proposed, such as *Invalidation Report (IR)* [5, 8–11, 18, 21–24, 26, 39, 45, 48, 51], *Poll-Each-Read* [25] and *Call-Back* [20, 25, 29, 48] policy.

The other aspect of a cache mechanism, the replacement policy, comes into play when the cache is full and an accessed object has to be accommodated. Here, one or more cached objects may have to be evicted to make room for the newly arrived object. Most research works have considered *Least Recently Used (LRU)* [5, 8–11, 18, 21–23, 25, 39, 45, 48, 51] or *Least Frequently Used (LFU)* [35] replacement policies for wireless data access. Note that, a cache mechanism may perform differently with different combinations of access and replacement policies, and therefore, a system developer has to be prudent in choosing the appropriate replacement and access policies.

1.4 Outline

The remainder of this thesis is organized into five chapters. Chapter 2 and 3 present the related research works, and the system model and the preliminaries for our problem, respectively. In Chapter 4 and 5 describes our research works on optimal cache policies and their evaluations. In the former chapter we concentrate on systems where update events take place at the server only. In contrast, in the later chapter, all update events are considered to take place at the clients. Finally, we conclude the thesis in Chapter 6, where we present the research issues we are currently investigating or intend to investigate in the future.

1.5 Thesis Contributions

In this thesis, we put forward a strongly consistent cache scheme for wireless clients scattered over a network spanning multiple wireless cells, where data updates may originate from any of the clients or the server. We strive to make three major contributions -

1. Firstly, we propose two strongly consistent cache access policies - *Proactive Access Policy* (PAP), and *Reactive Access Policy* (RAP).

2. Secondly, we introduce an replacement policy, called Update-oriented Replacement Policy (URP). Our access policies are designed keeping the replacement policy in mind. The access policies collect different access and update related information to facilitate the working requirements of the cache replacement policy. In turn, the replacement policy aims towards higher effective hits.

3. Thirdly, we analytically prove that our replacement policy ensures the optimal performance. As a result, this research provides the upper boundary for the worst case performance of any caching scheme and a foundation for average case performance comparison.

The design goals of the proposed cache mechanism are - (1) to increase the effective hit ratio, and (2) to reduce transmission cost (i.e., bandwidth consumption) by the applications. Extensive simulations are performed to validate our proposals and claims.

Chapter 2

Related Works

Many interesting research works have been done and published on caching in software systems. In this section, we present a brief study on the prominent works in this area.

2.1 ccache

The development of *ccache* evolved around the idea of compiler cache [41]. It is a software development tool that caches output of a C or C++ compilation process and saves the information so that the recurring instance of the same compilation process can be avoided and the results can be taken from the cache. In a development environment, it is a common practice for developers to do a clean build of a project. However, to perform a clean build, developers, at first, throw away all the information from the previous compilations – typically using the command `make clean`. By using the ccache tool, recompilation process becomes significantly faster. The basic idea is to detect re-compilation of the exact same code and then reuse the previously produced output. The detection is done by hashing different kinds of information that is unique for a compilation of a given code (often, in a file). The tool limits (user configurable) cache size, by simply removing older cached files.

2.2 Dynamic Programming

Dynamic programming is a very important category of computer algorithms [13]. Dynamic programming is applied to solve optimization problems where subproblems share sub-subproblems. In a variation of dynamic programming algorithms the concept of *memoization* is used to get rid of inefficiency of recurring computation of same subproblems. A memoized algorithm keeps a table consisting of an entry for each subproblem. The table

act as a computation cache. When a subproblem is encountered for the first time, a miss happens. The result for the subproblem is computed and stored in the corresponding table entry and the entry is marked in a way to indicate that the computation for the entry is done. Each recurring attempt of reevaluating the subproblem results in hits and simply needs to read the value from the related table entry. In a typical memoized dynamic programming algorithm, the table is considered to be large enough to hold solutions to all possible subproblems. Thus, cache eviction is not an issue here. Besides, as a table entry never changes after assigning a computed value, no cache coherency problem arises.

2.3 Harvest

Harvest [6] is a information discovery and access system. In Harvest, the difficulty of locating relevant information and bottlenecks of servers arising from popular content is addressed. Harvest uses an indexing scheme to search through crawled Web-pages, documents, and softwares available on the Web. To provide scalable access support, it adopts replications of indexes among hierarchically organized servers and caching of retrieved objects. This work assumes significantly large cache buffer which is, in this case, the secondary storages or disks. Simple purging of older object is used to make space for newer objects. Here, no consistency mechanism is required, as each replicated Harvest server act individually to serve their clients and no guarantee for the most updated information is provided. It is worth mentioning that Squid [38], the most famous Web proxy server till now, is an implementation of this research.

2.4 Adaptive Web Caching

The work on adaptive Web caching by Michel et al. is one of the earliest works to address the exponential growth of the Web [52]. This work was motivated by the performance limitation and administrative difficulties faced by Harvest (and, hence Squid) cache hierarchy and their administrators, respectively. Rather than a static hierarchy (as in Harvest), this works proposes to organize cache servers into a mesh represented by a multicast group. Thus, this work assumes availability of multicast as the method of efficient group organization and communication. Searching in between cache servers is performed by prefix matching of requested URL in a mapping table. The approach is similar to routing in IP networks, where packet is forwarder by prefix matching of network address in the routing table.

The work puts forward the idea of mesh organized Web cache servers, but it does not address the limitation arising from server capacities. For example, a server may not have

large storage to cache a smaller URL prefix, whereas storing a longer URL prefix may simply waste space. To better utilize cache space and for flexibility, a server may opt to host several longer URL prefixes. Furthermore, though significant research has been done on enabling multicast, till date, IP or other form of multicast has not been well adopted by industries or consumers.

2.5 Summary Cache

The collaboration among Web cache servers is severely constrained by the volume of information a server has to transmit to other servers to report the URL prefixes, the server is hosting. Moreover, caching longer URL prefixes results in larger URL prefix to server map, requiring longer search (i.e., CPU) time and more memory. To alleviate the problem, Fan et al. propose to use Bloom filter to share summery of a server content [17] with others.

In this solution, k independent hash functions, denoted as h_1, h_2, ..., h_k, are used. These hash functions map the input URL prefix to an integer in the range of 1 to m. The bloom filter is a simple bit vector of size m, initialized all bit with 0. For a given URL prefix A, $h_i(A)$-th bit of the filter is turned to 1 for $1 \leqslant i \leqslant k$. This hash processing is performed for all the cached URL prefixes. Instead of transmitting all the URL prefixes (or their MD5 [27] or similar hash table), a bloom filter is transmitted to inform collaborators about cached content of a server.

Searching in a Bloom filter starts with hashing a given URL prefix with the k hash functions and later checking all the bits positions returned by the hash functions. If all the bits are set to 1, the server, who created the filter, is considered to host the requested URL prefix. Though this solution of using Bloom filter never gives false negative results, but may suffer from false positives. It is shown that the probability of a false positive is $(1-(1-1/m)^n\, k)^k \approx \left(1-e^{kn/m}\right)^k$, where n is the number of URL prefixes in a filter. By carefully choosing m and k, the probability of false positive can be significantly reduced. It is shown that the approach reduces bandwidth consumption by 50% and CPU time by 30% to 95%.

2.6 EA-based Placement

Co-operative caching is often benefitted by the gain attained from the inter cache communication time over the fetch time from the server. Serving a missing objects from another cache in a collaborative way is the main focus for most of the researches. EA-based placement [34] is the first work to consider a placement scheme considering contentions at individual caches and limiting replications among a client group. Without proper placement

policy, an object is replicated in an uncontrolled way and all the caches in the group/system may possess replicas of the same object. Uncontrolled replication wastes disk space. With a proper placement scheme, greater number of different objects can be placed in the collaborative caches. This way the hit ratio is increased. In this research, a group of caches is proposed to be considered as a global resource and an expiration age-based estimation used to determine contention.

In this scheme, When a new object arrives, all existing objects whose sizes are equal to the new object are listed and LRU is applied on the list. If this list is empty, the process falls back to original LRU eviction policy. Besides, each object is assigned an expiration age at each client. The age is computed as the difference between the time the object is removed (i.e, evicted or moved to another cache) and last accessed, multiplied by the object size. Let denote this matric for the object D at client i as $EA(D^i)$. The average expiration age of the removed objects at a client within a predefined time windows is defined as the expiration age of a client. When a cache miss happens and the requested object is fetched from another client cache, among the two participants whoever has larger expiration age caches the objects to serve future requests. This way, the policy maintains only one copy of an object in the cache, and at the same time, strives to preserve objects for longer time.

An EA-based scheme does neither make any effort to increase utility of idle caches, nor try to preserve an object in the local cache until there is no other option but evict the object. Like other works, this Web-cache scheme does not utilize update information.

2.7 E-MACSC

The goal of E-MACSC [46] is different from most of the researches in the domain of caching. This scheme dynamically adjust cache size to match a given a hit ratio. The idea is to get the most utility out of the available memory while satisfying certain performance requirements. Access to online objects follow Zipf-like distribution and popularity distribution (PD) can be approximated with a straight line on a log scale plot. E-MACSC proposes to map this distribution to a bell shaped distribution by placing the most popular object at the center of the bell shape and alternate the other objects on two sides. With time, the depth of the curve representing the bell shaped distribution may change. After i-th sample, E-MACSC proposes to change cache size as follows:

$$CS^i = CS^{i-1} * \left(\frac{SD^i}{SD^{i-1}} \right)$$

where, CS^t and SD^t are the cache size after and standard deviation of object popularity during the t-th sampling period.

2.8 P2P Client Cache

In recent years, peer-to-peer (P2P) computing has been getting popular and the concept is applied to increasing number of applications, including caching. In [53], three kinds of cache organizations are considered. Firstly, a hierarchy of caches exists in an organization where the leafs of the hierarchy are the browser caches and rest are proxy cache servers. Secondly, the browser caches of an organization forms a network of P2P cache using an addressing technique known as Pastry [36]. Thirdly, the proxy servers forms a cooperative cache using Adaptive Web caching [52] or similar approach.

An object is first searched for browser cache, then in the Pastry network. If both fails, the request is forwarded to the local proxy server, which first try to locate the object within the organization and if that also fails, the object is searched in the cooperative network of the proxies.

This scheme uses a hierarchical greedy-dual approach, called $Hier - GD$, for replacement. If a new object is fetched at A, the following attempts are made to accommodate the object:

Step 1: If there is sufficient space at A, the object is stored locally.

Step 2: If previous step fails, use Pastry to determine the browser B, assigned to store the object. If B has sufficient space, store the object at B.

Step 3: If all above fail, remove the object with the lowest utility from A and store the new object locally.

Note that in the third step the lowest utility object is simply removed without considering possibility of relocation. A message is sent to the proxy server informing the identification of the browser finally hosting the new object. This way, other clients in the same organizational network and proxies from other organization can locate an object.

In this work, the reason behind emphasizing on storing an object locally is not clear. In a moderate to heavily busy system, the functionalities provided by the Pastry network are of no use, and all the requests for intra-organization hosted objects have to go through the proxy servers.

2.9 SPACE

SPACE [1] is a collaborative caching scheme for tightly coupled network stations to enhance filesystem performance. The authors argue that it is significantly faster to retrieve an object

from primary memory of a peer station, located in the same network, than to retrieve the object even from the local secondary storage. This work proposes to use bloom filter [17] to disseminate cache information among all the clients. This way, each peer cache is informed about contents of all other peers. When an object is not locally available, it is straight forward to find other peers hosting the same object.

When an individual cache becomes busy, this work proposes to utilize idle caches, and when a majority of the caches in the network becomes busy, duplicate copies of the same object are eliminated. To perceive this goal, when an object is fetched from the data server, the object is tagged as *original*. Replicas of the original objects do not have that tag. The working strategy of the scheme ensures that no more than one copy of an object is tagged as original[1]. To accommodate a new object, a cache tries the following steps in decreasing order of preference.

- If the cache has free buffer available, the new object is stored locally.
- The non-original object with the lowest utility is evicted to make space and the new object is stored.
- The original object with the lowest utility is evicted to make space and the new object is stored. This step is executed, if and only if the new object is a new object.

This scheme uses a slightly modifies LRU strategy to compute utility of a cached object. By evicting a non-original objects first, the scheme eliminates replicas of original objects from a busy cache. However, it gives higher chances to the original objects to survive in the network of caches. To prevent an individual busy cache from eliminating original objects, the scheme takes fewer more steps in decreasing order of preference:

- If another peer hosts a non-original replica of the to-be eliminated object, the other peer is requested to mark the replica as original.
- If there exits a peer with empty buffer, the original object is forwarded to that peer for preservation.
- If there exists a peer with non-original objects, the original object is forwarded to that peer with a request to evict one of the non-original objects and preserve the forwarded one.

[1]In fact, if more than one peer caches fetch the same object almost simultaneously, there may exist multiple original copies of the same object, but the scheme heals from this unwanted situation by itself within two protocol cycles.

To eliminate transmission of two bloom filters – one for original and one for non-original object, the scheme proposes to increase object ID space by one bit to indicate originality of an object. Then the content of a cache is encoded into a single bloom filter using the new address space and is broadcasted to other peers.

The scope of SPACE is limited to closely tight computers connected through high speed data networks. The sole goal is to increase file system performance through collaboration among caches. For updates, SPACE assume existence of a distributed consistency policy without considering the effects of updates on the cache contents.

2.10 Policies for Strong Consistency

For many data driven applications, exitance of more recently updated data, renders all older copy of the data invalid or useless. These application must always have access to the most recently updated data. This kind of consistency is known as the *latest value consistency* and the data satisfying the latest value consistency is known as *strongly consistent* [9]. In this section, we focus on representative works on latest value consistency policies, designed for wireless networks.

2.10.1 Invalidation Report

The policy of invalidation report (IR) has been utilized in several cache management schemes for mobile networks [5, 8–11, 18, 21–23, 39, 45, 48, 51]. In this policy, the server periodically broadcast an invalidation report in which the modified data objects are listed. Instead of querying the server directly about updated objects, the clients wait for the next IR message. After receiving an IR-message, a client invalidates all the updated objects. IR-based consistency policy is very scalable as a new client can simply join the network and wait for the next IR message to make the cache consistent.

In spite of its popularity, IR-based schemes suffer from some major constraints. First, these schemes have to endure long query response time. When an object is requested from the cache, the cache has no choice but wait for the next IR message to ensure validity of the cached copy. This way, the average time to response to a query is increased by half of the inter IR broadcast period. Second, an efficient implementation of IR-based scheme would require a cross layer support to achieve the desired performance. Cross layer support in network software architecture is yet to be available in consumer products. Third, IR-based cache schemes assume the wireless channel to be a broadcast channel and the server to be available locally. In today's ubiquitous wireless data access networks, the service area is divided into many cells and subscribers/users are distributed all over the service area. At

the same time, for practical and architectural reasons, the servers are located on a wired network, outside the control of the wireless service providers.

2.10.2 Poll-Each-Read (PER) and Call-Back (CB)

The Poll-Each-Read (PER) [25] and Call-Back (CB) [20, 25, 29, 48] policies for cache consistency are extended from the similar memory consistency policies in the domain of distributed computing and are tailored to suite for the wireless network applications. The main motivation behind these ideas is to get around the limitations of the IR-based policies. To conserve bandwidth, in both of these policies, an updated data object is not immediately delivered to the clients. In contrast, a valid object is forwarded only after receiving a request for the object from the client applications.

In PER policy, a client verify the validity of the cached object with the server at each access to the cached objects. If the server responds affirmatively, the object is served to the application. Otherwise, the most updated object is fetched and then served to the application. In CB policy, at each update of an object, the server broadcasts a message to all the clients (or in case of unavailability of an efficient broadcast method, unicast messages are sent only to the clients hosting copies of the updated object) requesting to invalidate all the cached copies. Whatever contents, a client cache hosts, are considered to be valid and is served to satisfy request from the applications without any further consultation with the server.

2.11 Update-based Cache Replacement

Cache schemes, utilizing PER and CB policy for consistency, mostly use LRU and LFU policies for cache replacement. Whenever a new object is fetched from the server and the local cache is full, one data object from the local cache is evicted and the new object is accommodated. Depending on the replacement policy, either the least recently used or the least frequently used object is evicted. However, both LFU and LRU policies do not consider invalidations of data objects while making eviction decision.

Whenever an object is updated, all the cached copies of the same object, located across the network, become obsolete. Evidently, frequency of update, as well as access should play key roles in deciding objects to evict. In [12], an update-based cache replacement policy is proposed. This policy emphasizes on access and update frequencies while making the eviction decision. The policy computes Access to Update Difference (AUD) of an object by deducting update frequency from access frequency. The object with the least AUD among the cached and the newly fetched object, is evicted.

To the best of our knowledge, this is the first work to acknowledge the effect of update on cache performance. Though the idea is interesting, it is not supported with any strong theoretical analysis. Moreover, the simulation results also show that the cache hit performance heavily depends on the pattern of access and updates.

Chapter 3

Preliminaries

We start this chapter with a description of our system model followed by sections presenting performance metrics and notations used in this thesis. Then, we present the preliminaries for the access policies. In the subsequent sections, we present the basic working steps for our proposed access and replacement policies. We close the chapter with a discussion on cache content invariants and a description of the simulation model used for evaluating the performance of the proposed policies.

3.1 System Model

Our system model is based on the wireless data access networks, already available in the consumer market [37]. In these networks, service areas are divided into a number of *location areas* (LA). An LA is further partitioned into a number of *cells*. Each cell has a *base station* (BS). Many *mobile stations* (MSes) reside in a service area and each of them connects to the closest BS. All the BSes within one LA are connected to a *mobile switching center* (MSC). All the MSCs are finally connected to the *public switched telephone network* (PSTN).

An example wireless data access network is shown in Fig. 3.1. An MSC (via a gateway) is connected to the Internet through either proprietary networks of the wireless career or through PSTN. For practical reasons, data servers of the online service providers are integrated to the Internet or to the service provider's network through wired infrastructure. As a result, any form of communication between a mobile device and a data server has to pass through the wireless section of the network, located between the mobile device and the corresponding BS. Notice that, existing 2G (such as, EDGE, CDMA2000 1xRTT), 3G (such as, UMTS, WCDMA, CDMA2000 1xEV-DO), 3.5G and 3.7G (such as, HSDPA, IEEE 802.16e), under deployment 3.9G (such as, Long Term Evolution (LTE)) and expected 4G

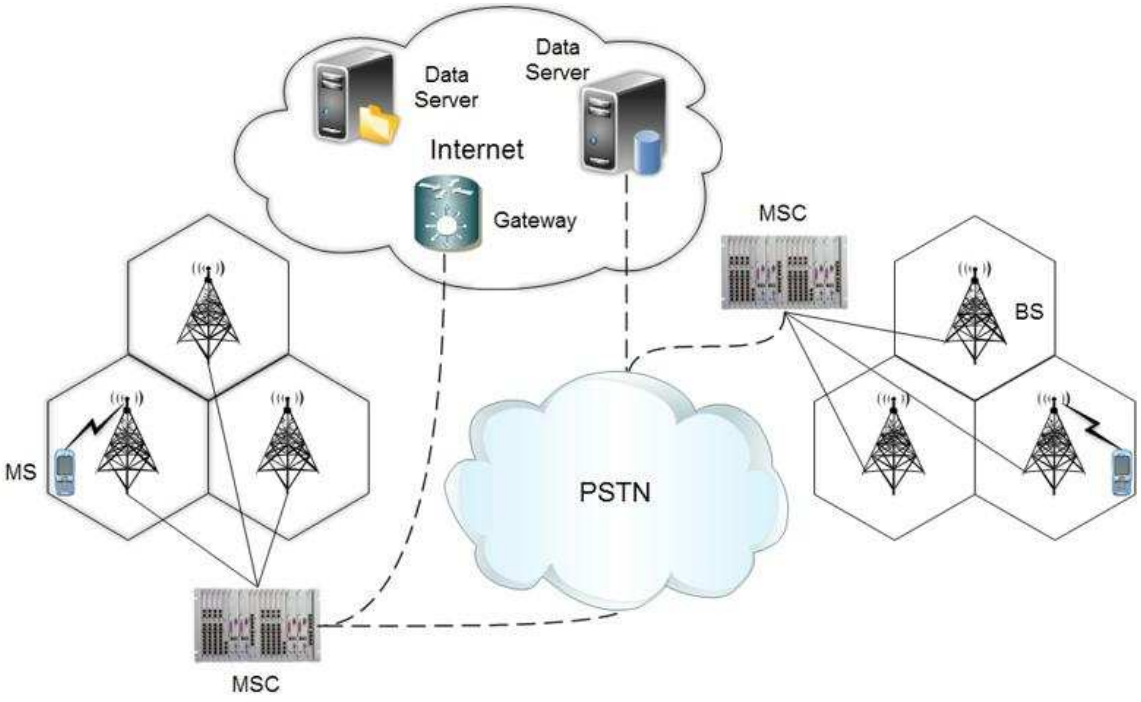


Figure 3.1: A Wireless Data Access Network

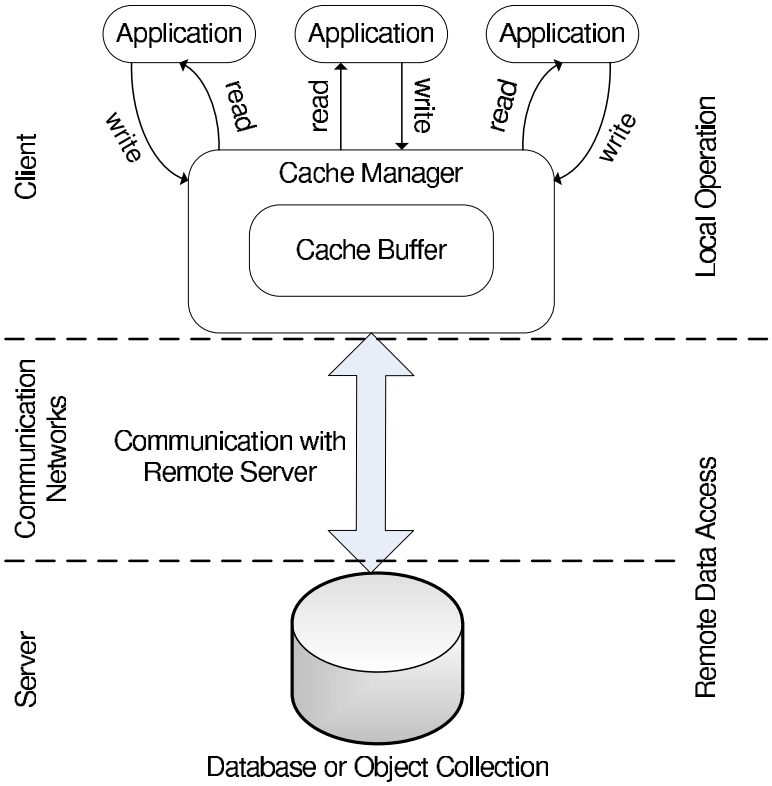


Figure 3.2: The Software Architecture

(such as LTE Advanced, IEEE 802.16m) networks are practical examples of our system model.

On their mobile devices (i.e., in mobile stations), users run different client applications. A group of interrelated data access applications may want to use a cache for the reasons described in the previous chapters. We consider applications for those strong consistency of data is a working requirement. A cache consists of a buffer and a cache manager[1]. In our model, a data server of a online service provider also plays an important role and collaborates with the cache managers in improving the performance of the client caches, scattered over the network (see Fig. 3.2). We made the design choice of leaving the network infrastructure of the wireless carrier unchanged. We are motivated by the following facts:

- As a mobile wireless device roams around from a cell to another, cache, maintained at a network infrastructure element, may have to be moved from one element to

[1]In the rest of the thesis, we use these terms mobile host, client (application), and user, interchangeably. Unless specified precisely, a cache and cache manager is considered to be an integral part of a client.

another. This process results in significant overhead and delay. Contrary, a cache at a mobile device is not influenced by the mobility of the user.

- Different online applications have different data semantics. It is really difficult to understand all the data semantics of fast evolving online applications.
- Beside data semantics, intermediate processing units may also have to understand the semantics of the communications between the clients and the server for efficient deployment of a cache. In many cases, these semantics are not even publicly available.
- A work around to the above problem is to allow the online service providers to add the correct data and communication semantics to the proper network components of the carriers. However, in most cases, online service providers and wireless carriers are different organizations, and online service providers do not have any control or access to the network components of the wireless carriers.
- Even if an understanding between online-service providers and carriers can be established (through proper standard interfaces), often communication between applications on mobile devices and online service providers are end-to-end encrypted. Thus, to allow the wireless carriers to understand the exact communication between the client and the online service providers, a new privacy and encryption model has to be investigated.
- Besides, a change to the wireless infrastructures is extra ordinarily expensive and often a very slow process to be universally adopted.

All accesses to the data objects, hosted by the data servers of the online service providers, take place at the mobile stations only[2]. Updates to the data object occur at two different places - (1) at the server, (2) at the clients. We assume that the access and update events are Poisson processes. In this thesis, we consider that communication scheduling, channel condition tracking, packet scheduling, error and flow control are performed by the lower layers of the communication stack and are the out of the scope of this thesis. Rather, we focus on reducing consumption of the amount of bandwidth a cache requests from the lower layers to serve a data object to the applications.

[2]Though the solution, proposed in this thesis, is equally implementable for non-mobile stations (i.e., wired part of the Internet), non-mobile stations are not counted towards the working steps of the solution. If non-mobile stations are also included in the solution, the entire system (including mobile and non-mobile stations) will ensure hit and bandwidth optimality, but may not be optimal for mobile stations only.

3.2 Performance Metrics

We have two major design goals - (1) increase the number of effective hits and (2) reduce the communication cost. To better understand these goals, we first define the following concepts. When an access takes place two situations may arise a *cache hit* or a *cache miss*. A cache miss happens when the accessed data object is not in the cache. Otherwise, a cache hit is considered to take place. However, not all cache hits contribute towards serving a data object from the cache. Cache hits are further classified into two groups - *valid cache hit* and *invalid cache hit*. A cached object becomes invalid when an updated version of the object is available. Invalid cache hits are due to invalid cached objects. Given that the cached object is the most recent version, a hit on the object results in a valid cache hit. *Effective hit ratio* is the ratio of a valid cache hit over all accesses.

According to our system model, the wireless channel bandwidth between the mobile devices and corresponding base stations is the most expensive resource. Hence, measuring *cost* involves the amount of average bandwidth, consumed by a cache to serve a requested data object to the applications.

3.3 Notations

Before describing the proposed policies, we introduce several notations to make the concepts clear. Let the number of distinct and equal size objects hosted by the server be N. The i-th hosted object is identified as O_i, where $i = 1, \ldots, N$. Let the maximum number of objects a client can locally cache be K. Let $\mu_i^j(t)$ and $\lambda_i^j(t)$ be the access and update rates, respectively, of O_i at client j up to time t. We denote μ_i^j and λ_i^j as the expected access and update rates at the client j for object O_i, respectively. With sufficiently large value of t, $\mu_i^j(t)$ and $\lambda_i^j(t)$ approach to expected access and update rate μ_i^j and λ_i^j, respectively. Formally, $\mu_i^j = \lim_{t\to\infty} \mu_i^j(t)$ and $\lambda_i^j = \lim_{t\to\infty} \lambda_i^j(t)$[3]. Let the rate of update to object O_i from the server be $\lambda_i^{\mathbb{S}}(t)$ and $\lambda_i^{\mathbb{S}}$ be the expected rate.

Let $\mu_i^{\mathbb{A}}(t)$ and $\lambda_i^{\mathbb{A}}(t)$ be the total access and update rate, respectively, for object O_i up to time t from all the clients. Expected access and update rate for object O_i over all clients are defined as $\mu_i^{\mathbb{A}} = \lim_{t\to\infty} \mu_i^{\mathbb{A}}(t)$ and $\lambda_i^{\mathbb{A}} = \lim_{t\to\infty} \lambda_i^{\mathbb{A}}(t)$, respectively. Obviously, $\mu_i^{\mathbb{A}}(t) = \sum_j \mu_i^j(t)$ and $\lambda_i^{\mathbb{A}}(t) = \sum_j \lambda_i^j(t)$. Let $\lambda_i(t) = \lambda_i^{\mathbb{A}}(t) + \lambda_i^{\mathbb{S}}(t)$ be the global update rate up to time t. Similarly, λ_i is defined. The global access rate up to time t, $\mu_i(t)$ is same as $\mu_i^{\mathbb{A}}(t)$. The expected metric μ_i is also defined accordingly. Similarly, $\mu^j(t)$, μ^j, $\lambda^j(t)$ and λ^j are defined as $\sum_i \mu_i^j(t)$, $\lim_{t\to\infty} \mu^j(t)$, $\sum_i \lambda_i^j(t)$, $\lim_{t\to\infty} \lambda^j(t)$, respectively.

[3]Depending on the characteristics of the concerned application, μ_i^j and λ_i^j are computed as statistical or time-based average.

3.4 Primitives for the Proposed Policies

Through the proposed access policies, we ensure that the objects, served to the applications, are the copies of the most recent version of the objects. Beside serving requested objects, these policies also handle updates injected by the clients or server, and collect access and update related information to facilitate working requirements of the replacement policy. Our access and replacement policies use following abstract[4] primitives:

add(obj): Add *obj* to the local cache. The prerequisite of this primitive is the availability of at least one free memory block in the cache buffer.

evict(id): Evict the object in the cache buffer with identification – *id*. The post condition of this primitive is one more free memory block in the cache buffer.

replace(obj): Overwrite an older copy of locally cached *obj* object with the most recent version of *obj*.

modify(prof): Depending on policies in place, access and update related profiles per object (and per client) are maintained. Each profile keeps access and/or update rates of the hosted objects and may contain information about availability of objects at different clients. This primitive modifies part of the profiles, indicated by *prof*.

find_replaced(rp): Find the identification of the object to evict, using the given replacement policy, identified by *rp*.

3.5 Working Steps of the Access Policies

In this section, we describe the working steps of our proposed access policies – (1) Reactive Access Policy (RAP) and (2) Proactive Access Policy (PAP). Like the PER and CB policies described in Chapter 2, RAP and PAP are also inspired by the memory consistency policies in distributed memory management.

In Reactive Access Policy (RAP)

In RAP, the clients are reactive in verifying the consistency of the associated data object at each access. In this policy, the server is the most resourceful and knowledgable entity in the system. It maintains per client per data object access and update profiles. It also keeps track of identities of objects, hosted by each client. The working steps of RAP is shown in Fig. 3.3.

[4]Efficient implementation details are left for the application designers and developers.

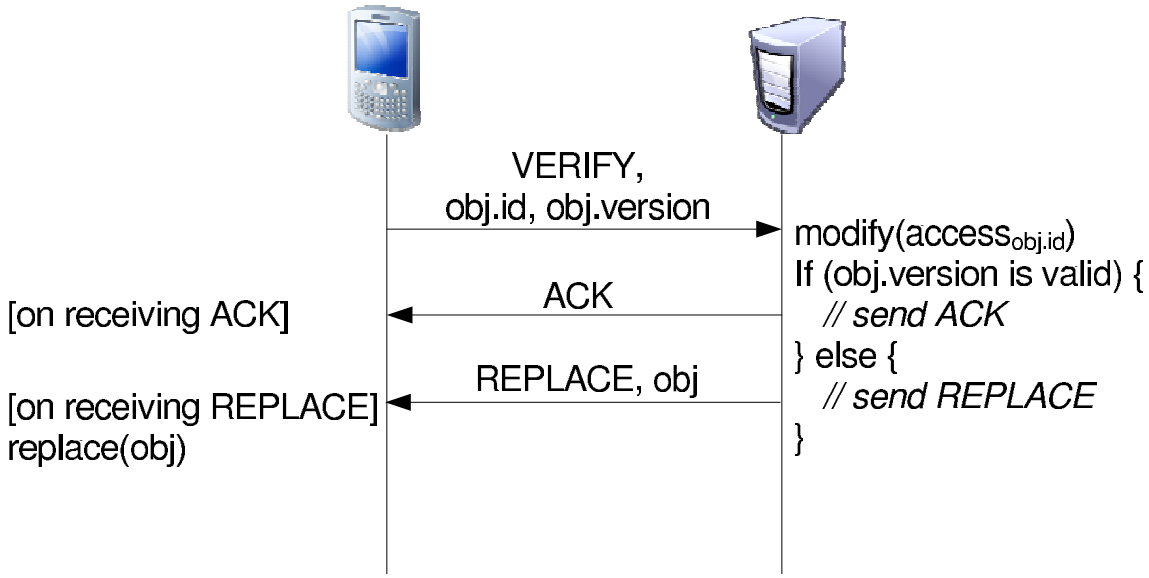
When a client makes a request for an object, in some cases, a copy of the object is available in the local cache. An event of access to such an object begins with a verification step, where the cache manager sends a query to the server (in a VERIFY message) requesting a check on validity of the cached copy. When the cached copy is consistent with the most recent version of the object, the server confirms the validity with an acknowledgement (as an ACK message). Otherwise, the server forwards a copy of the most recent version of the object (with a REPLACE message) and the cache manager replaces the older copy with the most recent version, received from the server. Fig. 3.3(a) illustrates the entire process.

In the other cases, no copy of the requested object is available in the local cache (as shown in Fig. 3.3(b)). Obviously, the client has to make a request to the server to fetch the object (in a REQUEST message). The server considers two different scenarios. First, if the client cache has enough buffer to accommodate at least one more object, the requested object is simply forwarded (in an ADD message) to client to be added to its cache buffer. Second, due to unavailability of sufficient space, a replacement decision has to be made. The server, the most knowledgeable entity in the system, makes the replacement decision and forwards the identification of the object to replace with a copy of the requested object (in an EVICT message).

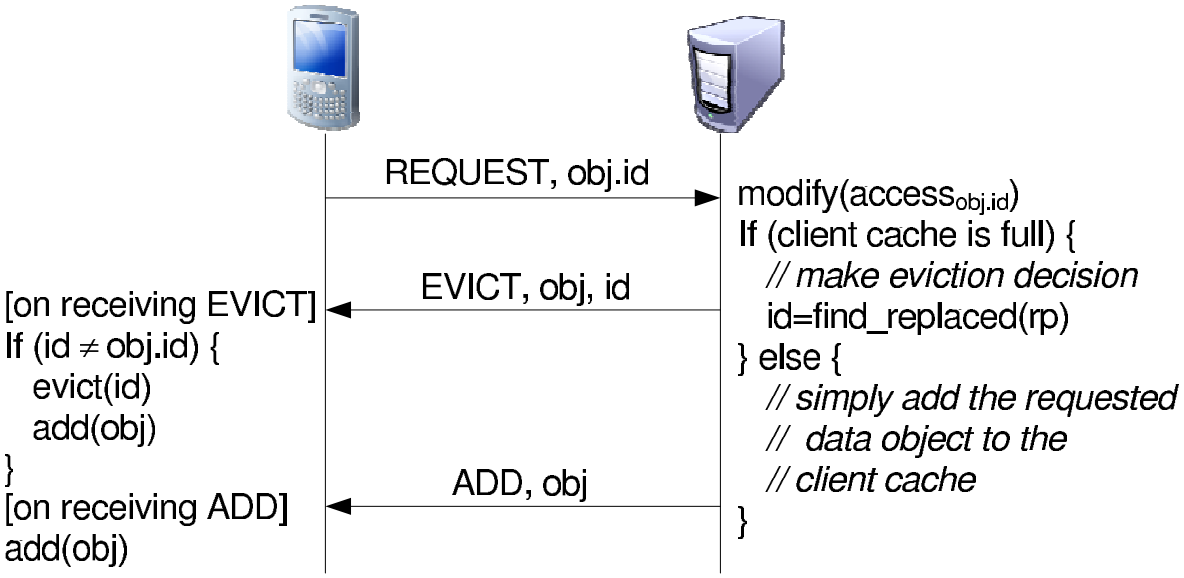
In Proactive Access Policy (PAP)

In PAP, as the server informs its clients about all the updates in proactive manner, each client precisely knows which locally cached objects are consistent. Thus, unlike RAP, at a request from an application, the cache manager does not need to consult the server to verify consistency. In this policy, the cache manager is responsible to make the replacement decision. To facilitate this decision making process, the manager maintains access and updates profile for all the cached objects. Note that, the update messages from the server can be transmitted as unicast messages. This form of communication is supported by all networks captured by our system model. However, the inefficiency of unicast messages can be overcome, if IP multicast is supported by the underlying networks. An alternate solution is to make the intermediate components (such as, gateways) of the network of the wireless carrier aware of the cache systems. This way, the intermediate components can take the advantage of supported broadcast channel and send a single message per affected cells for each update. Note that, involving intermediate components have other challenges, as described in Section 3.1.

When a consistent copy of a requested object resides in local cache, the object is served without any further processing. Otherwise, a sequence of steps are taken to satisfy the request (as shown in Fig. 3.4). A request to fetch the object is sent to the server (in a REQUEST message). The server fetches the object along with the relevant access and update profiles (in an ADD message). If the cache manager finds that the local cache



(a) Requested object is locally cached



(b) Requested object is not locally cached

Figure 3.3: Step sequences for RAP access policy

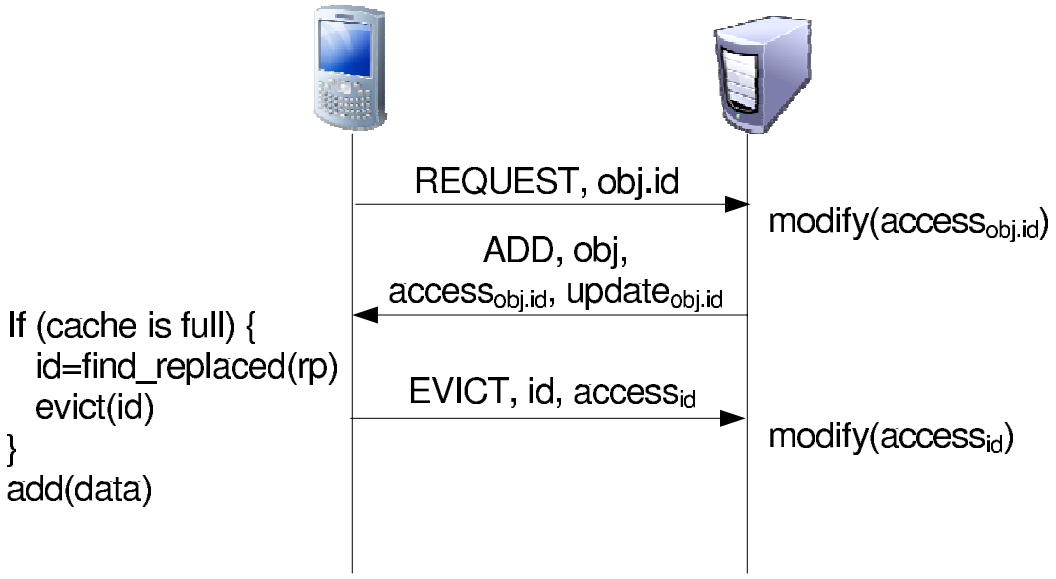


Figure 3.4: Step sequences for PAP access policy

is full, it makes a replacement decision before accommodating the newly received object. It must be noted that whenever the policy decides to replace an object (either due to replacement or update), the associated profiles are forwarded to the server.

3.6 Working Steps for Update-oriented Replacement Policy (URP)

The working steps for replacement (i.e., *find_replaced* primitive, as shown in Fig. 3.3 and 3.4) in URP is shown in Algorithm 1. The algorithm takes three inputs - (1) the gain factor computation policy (*compute_gf*), (2) the set of (attributes of the) currently cached objects (C), and (3) the accessed object (O_a). Note that, the last two inputs are implicit and hence omitted in primitive description or in Fig. 3.3 and 3.4. The gain factor of an object gives the utility of preserving an object in a cache. Computation of gain factor varies from one policy to another. This algorithm, at first, finds the object with the minimum gain factor (GF) (O_r) over all the cached and accessed objects (line $1-6$). If O_r and O_a are not the same (from Fig. 3.3 and 3.4), O_r is evicted from and O_a is saved to the cache. Otherwise, the cache is kept intact and O_a is not saved. Note that, we consider replacement and eviction to be different activities – replacement takes place whenever the cache is full and a new object is introduced to the cache. A replacement results in an eviction, only

when an in cache object is removed to accommodate with the newly introduced object. In summary, the replacement policy decides whether to replace an in cache object with a new object. If a replacement is necessary, the policy also decides about the in cache object to be replaced.

Algorithm 1: *find_replaced* - the replacement algorithm for client i

```
input  : compute_gf is the gain factor computation policy
input  : C is a reference to the set of currently cached objects
input  : O_a is the accessed object
// Cached object with the lowest GF
MinGF ← +∞;
foreach O_j ∈ (C ∪ {O_a}) do
    GF_i^j ← compute_gf(i, j);
    if MinGF > GF_i^j then
        MinGF ← GF_i^j;
        r ← j;
// r is the data object to be replaced
return r;
```

Note that Algorithm 1 is presented here just for better clarity. An implementation, may not follow the sequential operation shown here. Implementation wise, the proposed cache scheme may have similar data structures and operations of the popular LFU scheme. A simple implementation would maintain the cache content and other related information in a linked-list and would evaluate Algorithm 1 at each replacement. In another implementation, the cache content is maintained as a min-priority queue where priority of a cache content is same as the GF value of the content. However, as GF value changes at each replacement the priority queue also needs to readjusted. To facilitate faster search in cache, a variant of binary search tree (BST) can be implemented with proper reference to the cache content. At each access, a search in the BST takes $O(n)$ time. The former way to implement cache content incurs no additional cost at the access, but the later way incurs $O(1)$ additional operation[5]. A deletion of an object (at an eviction or invalidation) would require $O(lgn)$ time to rearrange the BST. The former implementation would need $O(n)$ and later $O(lgn)$ additional operations. Finally, to find the object to evict, the former implementation would take $O(n)$ and the later $O(lgn)$ operations.

[5]We assume Fibonacci Heap as the choice to implement priority queue among many others. We also assume that *compute_gf* is a constant operation. In later chapters, we will find that *compute_gf* is in deed a constant operation.

3.7 Cache Content Invariants

During a cache access, if the accessed object is not available at the local cache, the object is fetched from the server. When the local cache is full and a new object is introduced (due to access), a replacement decision has to be made. The employed replacement policy makes the final decision about preserving the new object. In case the new object is preserved, another object from the local cache is evicted to make space for the new object. To facilitate the analysis, let us number all replacements in the sequence they take place in time. The earliest replacement is identified as the first replacement, the next one as the second replacement and so on. Let denote the sets consisting of all cached objects before and after the t-th replacement with $C(t)$ and $C'(t)$, respectively.

Definition: The *guaranteed effective hits* is the worst case effective hits resulting from cache hits to the in cache objects only.

An update may invalidate in cache objects and create free space in the cache buffer. This way, due to availability of free buffer, a cache may accommodate a new object after a cache miss without evicting any in cache object. In the worst case, these new objects have such access and update patterns that they do not contribute towards increasing the number of effective cache hits. Even though these objects are not proving any utility, they are accommodated in the cache, because of the available space.

Let the probability of guaranteed effective hits after the t-th replacement for RAP and PAP be $P_{RAP}(t)$ and $P_{PAP}(t)$, respectively, under an arbitrary replacement policy. Similarly, the expected cost of accessing an object resulting from the t-th replacement are denoted as $C_{RAP}(t)$ and $C_{PAP}(t)$, accordingly, under any replacement policy. We use $P_{RAP+URP}(t)$, $P_{PAP+URP}(t)$, $C_{RAP+URP}(t)$ and $C_{PAP+URP}(t)$ to denote the relevant metrics due to the t-th replacement when URP is exercised. Let $O_a(t)$ and $O_r(t)$ denote the accessed and replaced objects at the t-th access, respectively. Let $O_{URP}(t)$ denote the replaced object when URP is used. The cost of transmission of an object, a request (or verification) message, and an acknowledge message are denoted with the notations C_{obj}, C_{req}, and C_{ack}, respectively. In this research, we leave out the costs of transmitting replacement decisions or cost of piggy backing profile information with a large message, such a message with a data object. These information incurs very negligible message overheads.

The following invariants hold at the t-th replacement event,

$$O_a(t) \notin C(t) \tag{3.1}$$

$$O_r(t) \in C(t) \cup \{O_a(t)\} \tag{3.2}$$

$$C'(t) = (C(t) \cup \{O_a(t)\}) \setminus \{O_r(t)\} \tag{3.3}$$

Similarly, for URP policy,

$$O_{URP}(t) \in C(t) \cup \{O_a(t)\} \tag{3.4}$$

$$C'(t) = (C(t) \cup \{O_a(t)\}) \setminus \{O_{URP}(t)\} \tag{3.5}$$

3.8 Simulation Model for Performance Evaluation

To evaluate the performance of our proposed policies, we have developed a discrete event simulator in written in C++. Based on the prior discussion, we have also evaluated the performance of the access policies combined with the popular Least Frequently Used (LFU) replacement policy [7]. We have collected information related to different performance metrics, discussed in subsection 3.8.1, from the simulations and summarized in subsequent parts of this thesis. Our simulation environment is described in subsection 3.8.2. When we present simulation results, if applicable, we simultaneously present results from theoretical analysis.

3.8.1 Performance Metrics

We consider two performance metrics - effective hits and cost per access. Let $n_{a,j}$ be the total number of accesses to object O_j from all the clients. Let n_{miss} denote the total number of cache misses and invalid cache hits. Let n_a be the total number of accesses to all the objects from all the clients, i.e, $n_a = \sum_{j=1}^{N} n_{a,j}$. We compute the effective hit ratios as follows:

$$P_{RAP} = 1 - \frac{n_{miss}}{n_a} = P_{PAP} \tag{3.6}$$

In our simulations, we don't compute the cost of piggy backing information. For example, when access or update rates are sent with other large message load (such as an object), the cost of transmitting access or update rates is ignored. We also keep aside the cost of forwarding updates from the clients, because irrespective of the choice of the access policy,

this cost is fixed. Finally, the costs per access for RAP and PAP are computed according to (3.7) and (3.8), respectively.

$$C_{RAP} = \frac{1}{n_a}[(n_a - n_{miss}) \times (C_{req} + C_{ack}) + n_{miss} \times (C_{req} + C_{obj})] \tag{3.7}$$

$$C_{PAP} = \frac{1}{n_a}[n_{miss}(C_{req} + C_{obj})] \tag{3.8}$$

3.8.2 Simulation Setup

In data access applications, popularity of different objects are different. Researches have shown that user interest in different online objects follow *Zipf-like* distributions [7, 25]. In our simulations, we assume *Zipf-like* distributions for object access or update pattern, and at an access or update event, an object with rank i is accessed or updated with the probability p_i, defined as,

$$p_i = [i^{\alpha}(\sum_{j=1}^{N} \frac{1}{j^{\alpha}})]^{-1}$$

where $\alpha \geqslant 0$ and is called the *Zipf ratio*. Note that, when $\alpha = 0$, $p_i = 1/N$, for all i, and all objects are chosen with the same probability of $1/N$. Let α_a and α_u be the Zipf ratio for access and update events, respectively. We uniquely rank each object within the range from 1 to N to find its access and update probability. Note that rank of the object O_i may not be related to the object identifier i, as well, rank for access and update may be distinct. By default, we consider that C_{req} and C_{ack} have the same value of C_{msg}. Unless mentioned otherwise, the value of C_{msg}, C_{obj} and μ are 60, 600, and 1, respectively. Note that, if transmitting an object is not much costlier than transmitting a simple message, using cache is not worthwhile. In fact, in such case, using cache results in more communication and computation cost and unnecessary space (for cache buffer) consumption. We consider that 20 mobile users are in our network.

Chapter 4

Server Injected Updates

In this chapter, we discuss the access and replacement policies for applications where the updates are injected from the server only. In the first two sections, we discuss the steps for the access and update policies. Optimality of a caching scheme with these access and update policies is proven in the next section. We end this chapter by presenting our simulation results.

4.1 The Update Process

PAP and RAP policy differs in the process of notifying the clients about the update events. PAP follows a proactive approach for notification, where as, RAP is reactive. Whenever the server receives an update, in PAP, the server notifies all the clients hosting a copy of the same object about the availability of a newer version. This way, a client is able to remove locally cached invalid objects. Notice that in order to notify the clients about the update events, the server has to maintain a profile indicating cache contents of all its clients. In contrast, in RAP, a client checks for consistency of a requested object with the server in proactive manner and the server simply waits for such queries.

4.2 The Update-oriented Replacement Policy (URP)

In this subsection, we present Update-oriented Replacement Policy (URP) for cache system with updates injected from the server only. The policy uses both update and access frequencies gathered through the access policies to achieve superior guaranteed performance. We use *gain factor (GF)* of an object to determine which object should be given preference

in preserving in the cache. We define GF for the object O_i at client j up to time t according to (4.1). The logic behind this definition of GF is elaborated in the next section.

$$GF_i^j(t) = \frac{(\mu_i^j(t))^2}{\mu_i^j(t) + \lambda_i^{\mathbb{S}}(t)} \tag{4.1}$$

Similarly, long term GF is defined as,

$$GF_i^j = \lim_{t\to\infty} GF_i^j(t) = \lim_{t\to\infty} \frac{(\mu_i^j(t))^2}{\mu_i^j(t) + \lambda_i^{\mathbb{S}}(t)} = \frac{(\mu_i^j)^2}{\mu_i^j + \lambda_i^{\mathbb{S}}} \tag{4.2}$$

The algorithmic form of GF computation ($gf\_server\_injected$) is shown in Algorithm 2. In both PAP and RAP, as shown in Fig. 3.3 and 3.4, and in Algorithm 1, $gf\_server\_injected$ is considered to be the argument to the $find\_replaced$ primitive.

Algorithm 2: *gf_server_injected* - GF Computation in algorithmic form

input : i is the objects identifier
input : j is the client identifier
output: Gain factor for requested object i at client j
return $\frac{(\mu_i^j)^2}{\mu_i^j+\lambda_i^{\mathbb{S}}}$;

4.3 Quantitative Analysis

In this section, we analyze cache schemes with PAP and RAP for data access, and URP for replacement. Based on the behavior of URP policy, we conclude the following theorem.

Theorem 1 *URP policy maximizes the probability of guaranteed effective hits at each replacement when either PAP or RAP is employed for data access and all updates are injected by the server.*

Proof: To prove the theorem, we must show that the probability of guaranteed effective hits after t-th replacement obtained by replacing $O_{URP}(t)$ with $O_a(t)$ is larger than or equal to the same probability obtained by replacing $O_r(t)$ with $O_a(t)$.

Since access event is a Poison process, at any access, the probability of the accessed object being a given object O_i, $1 \leqslant i \leqslant N$, at client j is

$$p_{a,i}^j = \frac{\mu_i^j}{\sum_i \mu_i^j} = \frac{\mu_i^j}{\mu^j} \tag{4.3}$$

The probability of O_i being accessed before an update taking place is[1],

$$\overline{p^j_{u,i}} = \frac{\mu^j_i}{\mu^j_i + \lambda^{\mathbb{S}}_i} \tag{4.4}$$

Note that $\overline{p^j_{u,i}}$ also represents the probability that O_i is locally accesses before the object become invalid. Therefore, from (3.3), (4.3), and (4.4), no matter what replacement policy is used, we have,

$$\begin{aligned}
P_{PAP}(t) &= P_{RAP}(t) \\
&= \sum_{\forall i | O_i \in C'(t)} p^j_{a,i} \overline{p^j_{u,i}} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} p^j_{a,i} \overline{p^j_{u,i}} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} \frac{\mu^j_i}{\mu^j} \frac{\mu^j_i}{\mu^j_i + \lambda^{\mathbb{S}}_i} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} \frac{(\mu^j_i)^2}{\mu^j(\mu^j_i + \lambda^{\mathbb{S}}_i)} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu^j_i)^2}{\mu^j(\mu^j_i + \lambda^{\mathbb{S}}_i)} - \frac{(\mu^j_r)^2}{\mu^j(\mu^j_r + \lambda^{\mathbb{S}}_r)}
\end{aligned} \tag{4.5}$$

So, the equality when the URP is exercised can be driven as follow:

$$P_{PAP+URP}(t) = P_{RAP+URP}(t) = \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu^j_i)^2}{\mu^j(\mu^j_i + \lambda^{\mathbb{S}}_i)} - \frac{(\mu^j_{URP})^2}{\mu^j(\mu^j_{URP} + \lambda^{\mathbb{S}}_{URP})} \tag{4.6}$$

According to the definition of the URP policy when all updates are injected only by the server, we have,

$$\frac{(\mu^j_{URP})^2}{\mu^j_{URP} + \lambda^{\mathbb{S}}_{URP}} \equiv \min_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu^j_i)^2}{\mu^j_i + \lambda^{\mathbb{S}}_i} \leqslant \frac{(\mu^j_r)^2}{\mu^j_r + \lambda^{\mathbb{S}}_r} \tag{4.7}$$

[1]The detail derivation of the equation is shown in Appendix E

Therefore, we conclude,

$$P_{PAP+URP}(t) = P_{RAP+URP}(t) = \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu_i^j)^2}{\mu^j(\mu_i^j + \lambda_i^{\mathbb{S}})} - \min_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu_i^j)^2}{\mu^j(\mu_i^j + \lambda_i^{\mathbb{S}})}$$
$$\geqslant \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu_i^j)^2}{\mu^j(\mu_i^j + \lambda_i^{\mathbb{S}})} - \frac{(\mu_r^j)^2}{\mu^j(\mu_r^j + \lambda_r^{\mathbb{S}})} = P_{PAP}(t) = P_{RAP}(t) \tag{4.8}$$

From the above derivation, it is conspicuous that at each replacement, the probability of effective hits while using URP is greater than or equal to the same metric while using any other replacement policy. Thus, we conclude that URP maximizes the probability of guaranteed effective hits at each replacement for both RAP and PAP access mechanism when all the updates are injected from the server.■

Corollary 1 *In a cache system when updates are injected only from the server, URP minimizes the expected cost of data access at each replacement for both RAP and PAP access mechanism.*

The corollary is proved in Appendix B. Based on the characteristics of URP discussed above, we further propose the following Theorem:

Theorem 2 *In a cache system if all the updates are injected only from the server, in the long run, URP guarantees optimal effective cache hits for both RAP and PAP.*

Detail proof is presented in Appendix C. We can also assert that,

Corollary 2 *In a cache system when updates are injected only from the server, URP minimizes the expected cost of data access in the long run for both RAP and PAP access mechanism.*

Proof of the corollary is presented in Appendix D.

4.4 Performance Evaluation

In this section, we present different results available from our extensive simulations. Our simulation environment has already been described in Section 3.8.

4.4.1 Objects with No updates

Fig. 4.1-4.3 show the effective hits and expected cost for URP and LFU scheme when no update to any object take place, i.e., $\lambda_i^{\mathbb{S}}$ for all i is 0. In all the simulations, the server is populated with 500 objects. In this case, URP_i^j is determined by μ_i^j and consequently, while accommodating new objects in the cache, both URP and LFU choose the same objects for eviction. Thus both the policies result in same performance. In addition, we have the following observations:

• All combinations of access and replacement policies enjoy higher number of effective hits and lower cost when the objects are for read only and no update takes place (i.e., $\lambda = 0$). Static databases, audio and video files sharing in wireless environment are examples of this kind of application.

• LFU performance is also optimal where no update takes place in the caching system.

• With increment of α_a, number of effective hits also increases. The higher α_a is, the smaller set of objects is accessed more frequently, resulting in fewer misses.

• Larger cache helps alleviating cache misses. However, this behavior is clearly visible when α_a is smaller. With larger α_a, most of the access are due to fewer objects and hence, increasing cache size results in very little additional benefit.

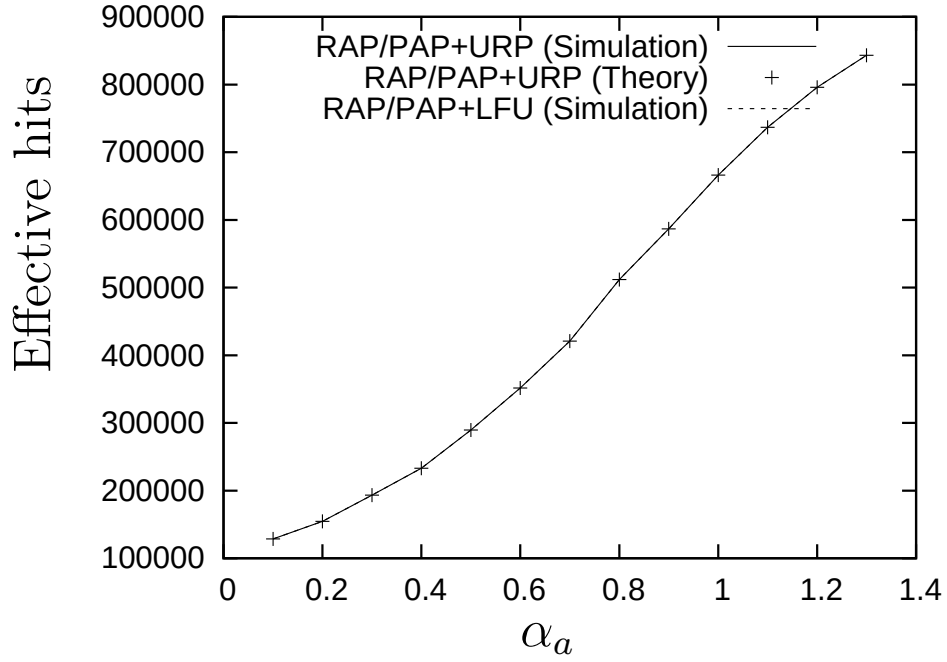
4.4.2 Impact of Number of Objects and Cache Size

In this subsection, we present the effect of number of objects (N) and cache size (K) on cache performance. In Fig. 4.4 and 4.5, the performance of PAP/RAP+URP is compared with PAP/RAP+LFU for different object set sizes. The values for the parameters α_a, α_u and K in these simulations are 0.20, 0.60 and 20, respectively. We have the following observations:

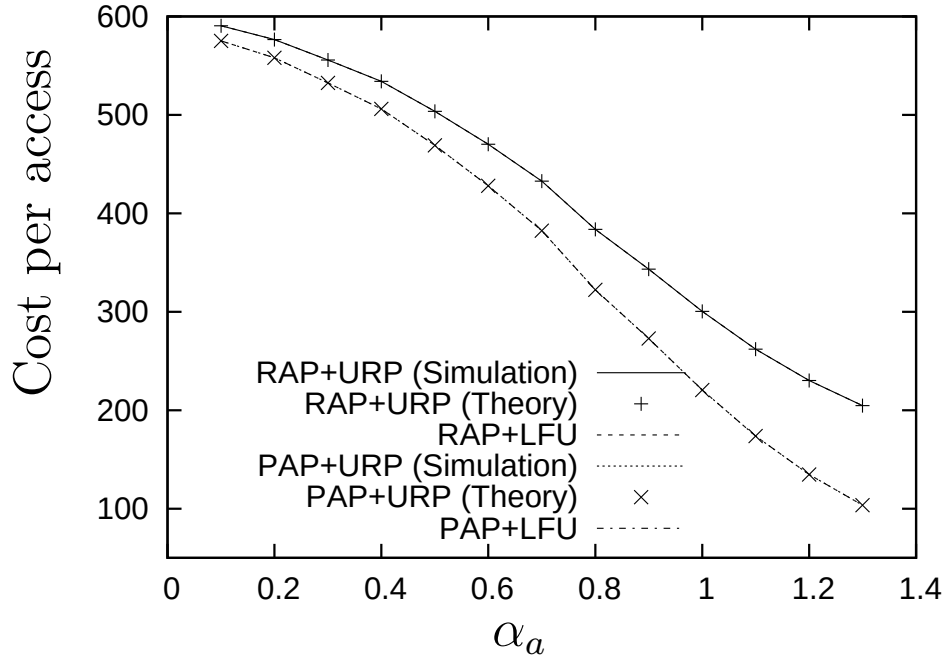
• Number of effective hits for all combinations of access and replacement policies decreases with the increment of database size. However, in all cases, URP replacement policy shows better results.

• Given a fixed size cache, as the object set size increases, the chance of hit reduces, irrespective of the replacement policy. However, the gain of using URP becomes prominent with larger set sizes.

• PAP+URP policy gives the best performance in terms of cost per access and PAP+LFU closely follows. Provided that $C_{obj} \gg C_{msg}$, according to (3.7), the difference between PAP+URP and PAP+LFU is dominated by n_{miss}. Hence, with smaller miss rate, the gap between URP and LFU policies dilutes.

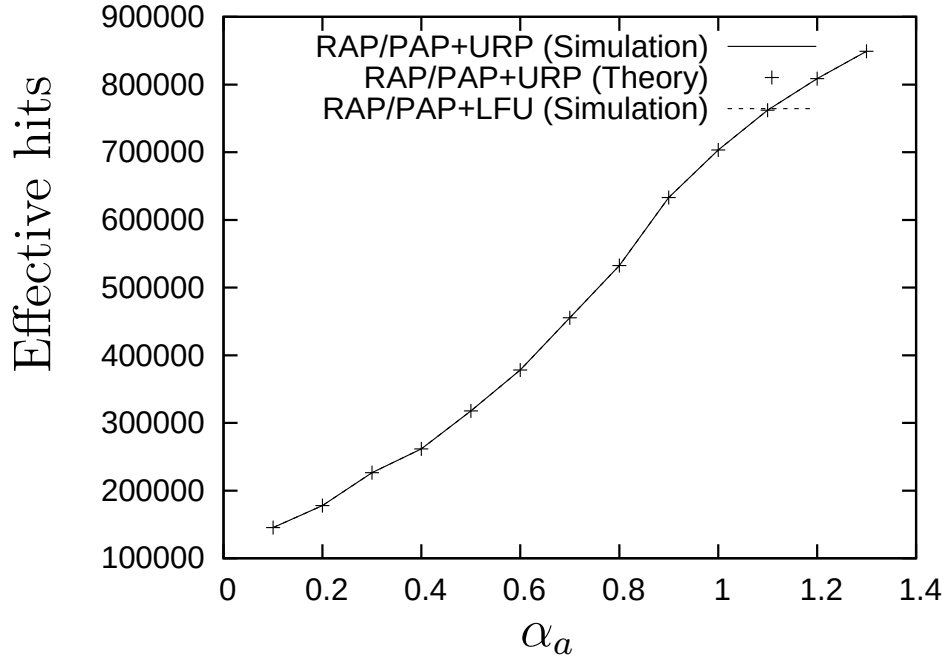


(a) Number of Effective Hits

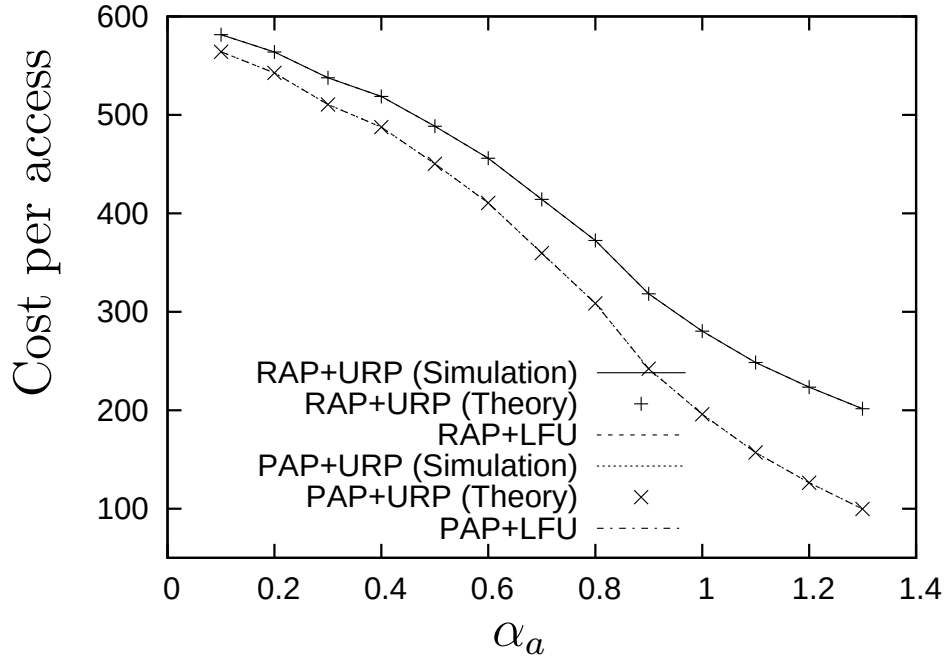


(b) Communication Cost

Figure 4.1: Performance of URP and LFU for readonly objects ($K = 50$)

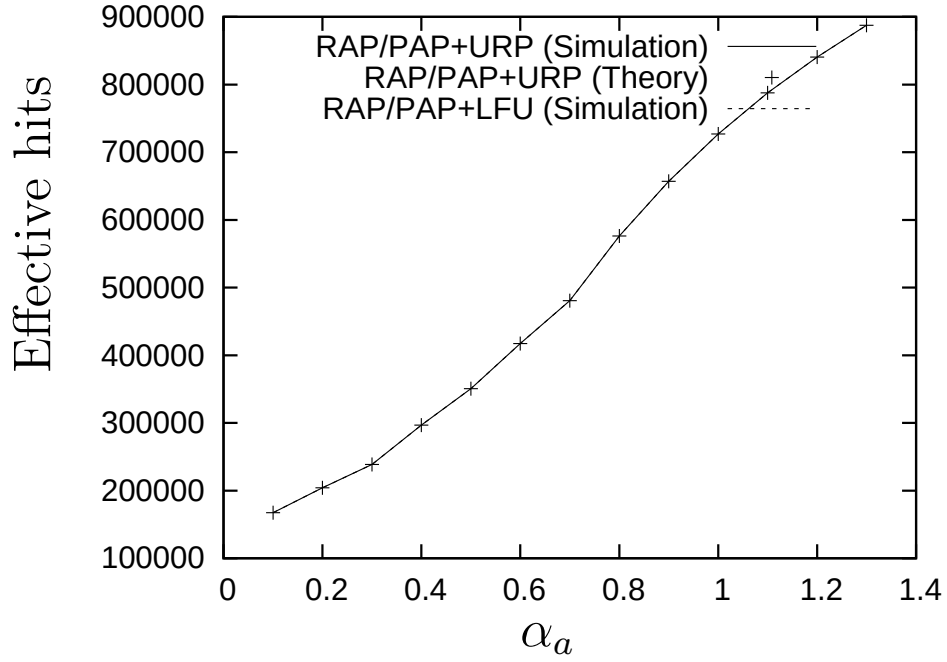


(a) Number of Effective Hits

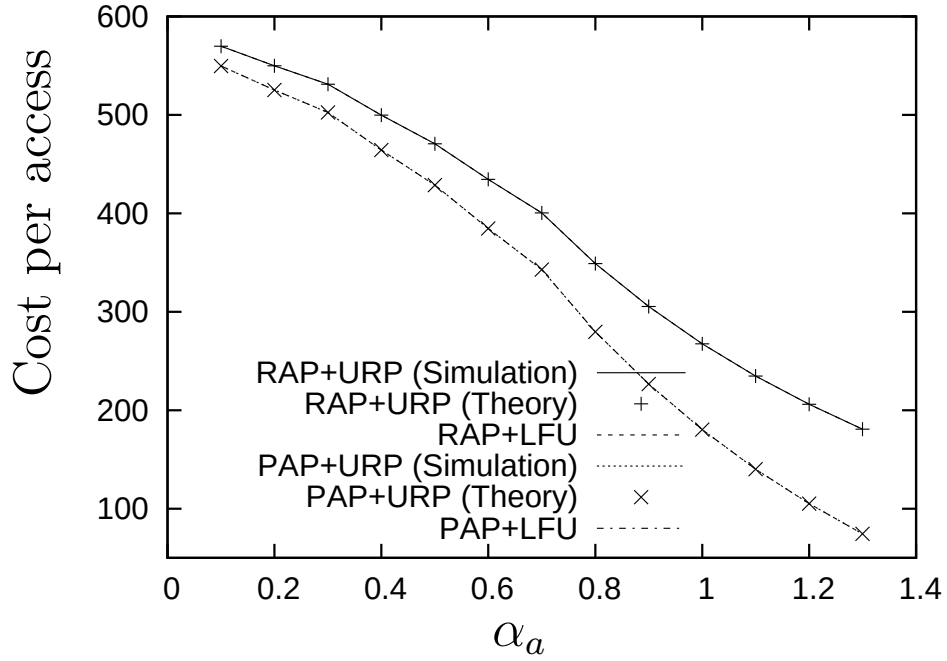


(b) Communication Cost

Figure 4.2: Performance of URP and LFU for readonly objects ($K = 60$)



(a) Number of Effective Hits



(b) Communication Cost

Figure 4.3: Performance of URP and LFU for readonly objects ($K = 70$)

• With any of the replacement policies, PAP access policies cost less than RAP. However, URP policies reduce cost per access further as compared to LFU.

Fig. 4.6 and 4.7 shows different performance characteristics for different cache sizes. For these simulations, the values for the parameters α_a, α_u and N are chosen as 0.01, 0.60 and 400, respectively. From the figures, we deduce following arguments:

• In opposed to the previous case, as cache size increases, the number of effective hits also increases for all combinations of policies. However, like the previous case, URP performs better in all cases.

• Due to increasing number of hits, the cost per access also declines with larger cache for both the policies. However, the cost of PAP is reduced at a higher rate than that's of RAP.

• As cache size increases, each additional extra cache buffer adds fewer additional effective hits. Hence, a designer must compromise between the cost of adding extra cache buffer and cache performance.

4.4.3 Impact of Zipf Ratio

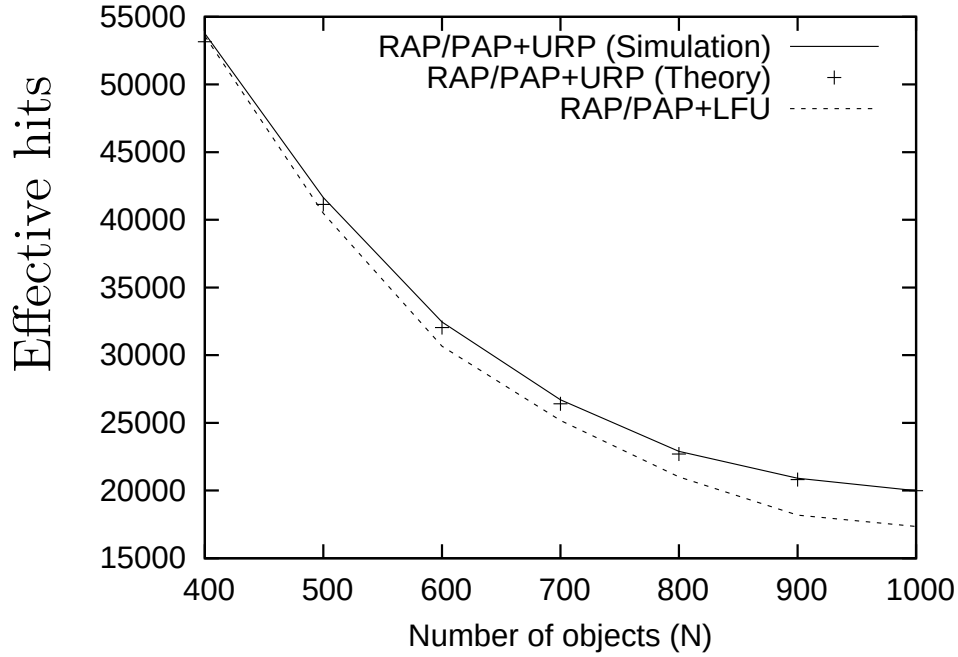
Fig. 4.8 and 4.9 show the effect of update Zipf ratio on the performance of both URP and LFU policies with the simulation parameters α_a, K and N set to be 0.01, 50 and 500, respectively. We observe that:

• With different Zipf ratios, performance of URP is consistently better than that of LFU in terms of both number of hits and cost per access.

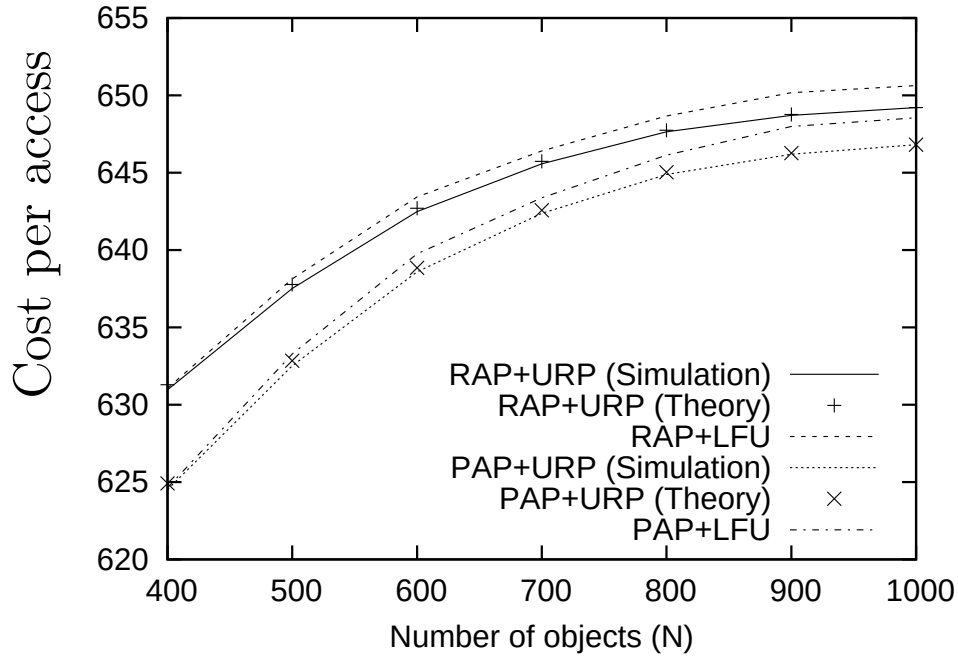
• The over all effective hits falls and cost increases as update Zipf ratios increases.

4.4.4 Impact of Number of Mobile Stations

Number of mobile stations has significant effect on what GF values different objects get. In the way GF is computed (as shown in (4.2)), $\lambda_j^{\mathbb{S}}$ is a global metric and is not affected by the number of mobile stations. On the other hand, μ_i^j for all i are affected by the number of mobile stations, provided that $\mu_i^{\mathbb{A}}$ for the entire system is given. As the number of mobile stations increases, the difference between GFs of different items become less distinct and GFs are mainly determined by λ. An update event has more severe adverse effect as the number of mobile stations increases. Fig 4.10 and 4.11 shows results with parameters α_a, α_u, K, and N valued at 0.01, 0.06, 50, and 500, respectively. As shown in the figures, with more MSes the hit rate decreases at a faster rate and hence, cost per access increases.

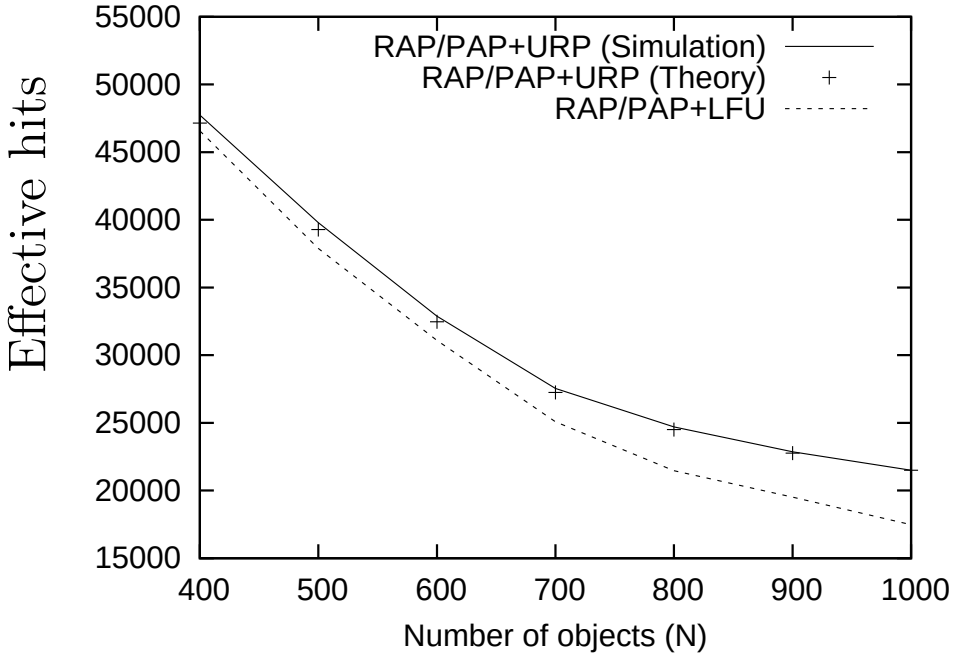


(a) Number of Effective Hits

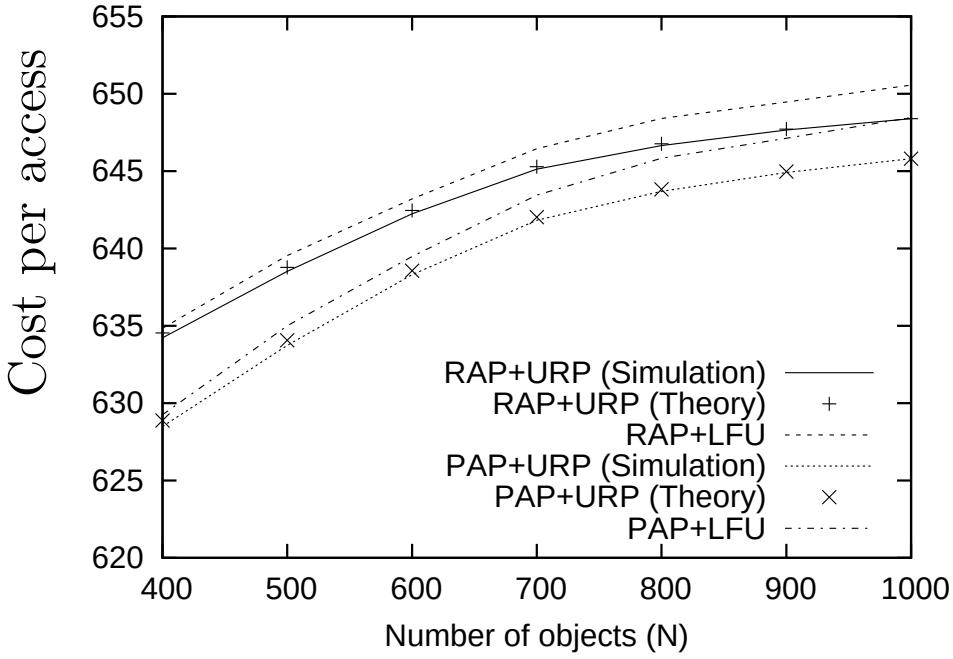


(b) Communication Cost

Figure 4.4: Performance of URP and LFU for different number of objects ($\lambda = 0.40$)

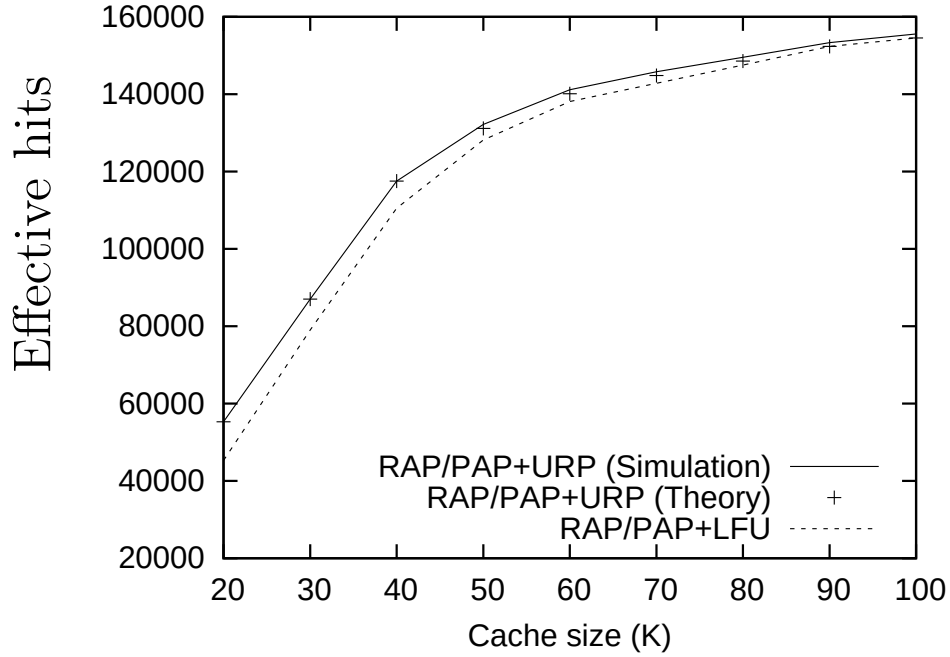


(a) Number of Effective Hits

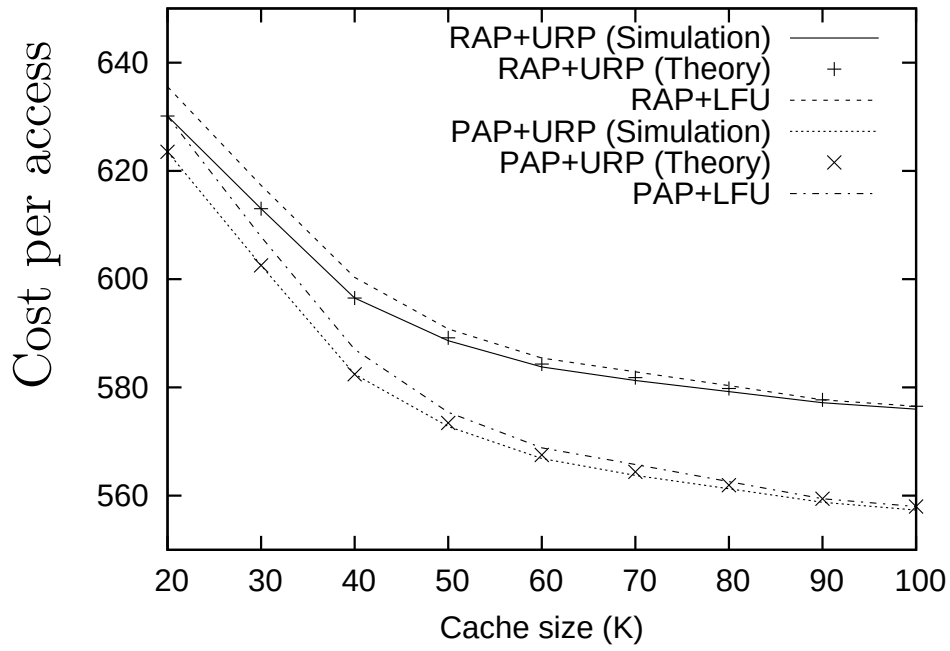


(b) Communication Cost

Figure 4.5: Performance of URP and LFU for different number of objects ($\lambda = 0.60$)

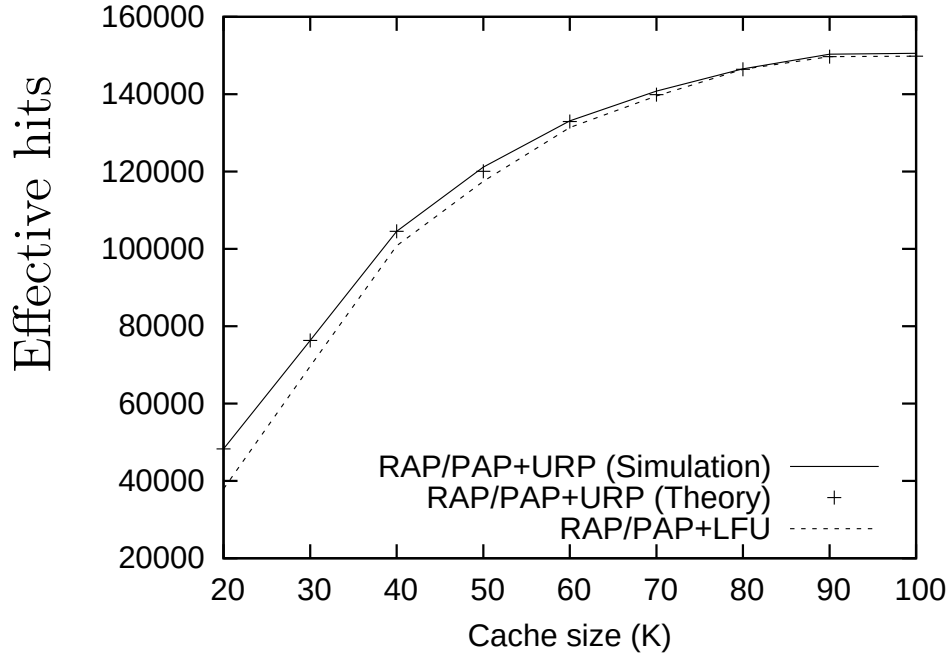


(a) Number of Effective Hits

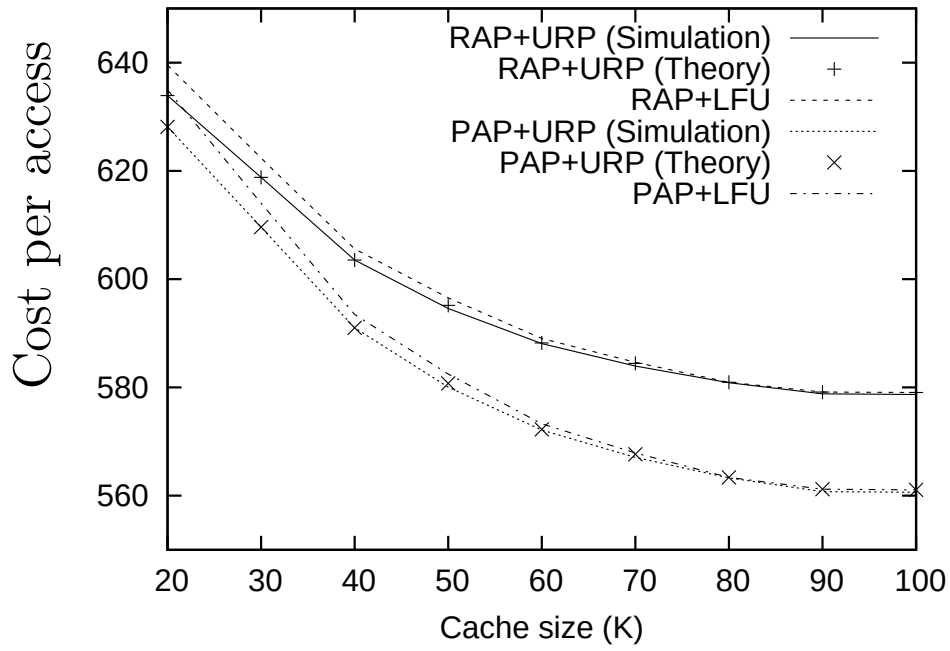


(b) Communication Cost

Figure 4.6: Performance of URP and LFU for different cache sizes ($\lambda = 0.50$)

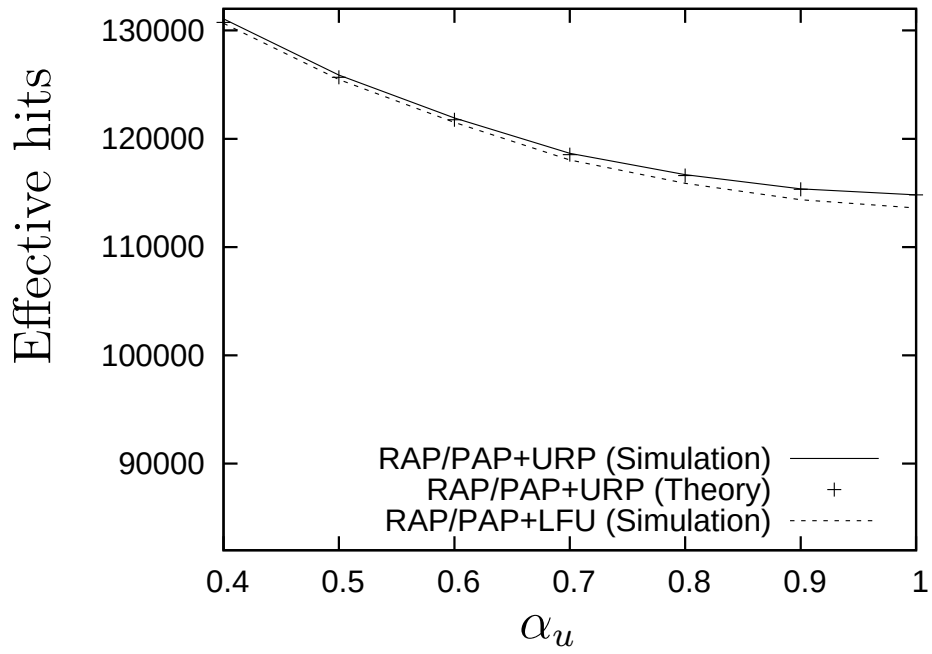


(a) Number of Effective Hits

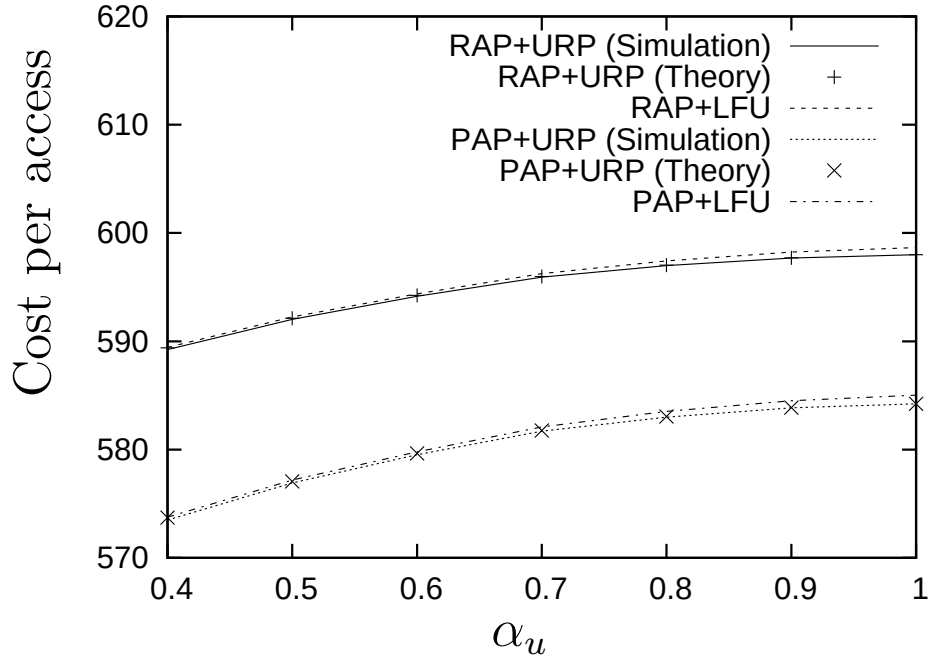


(b) Communication Cost

Figure 4.7: Performance of URP and LFU for different cache sizes ($\lambda = 0.60$)

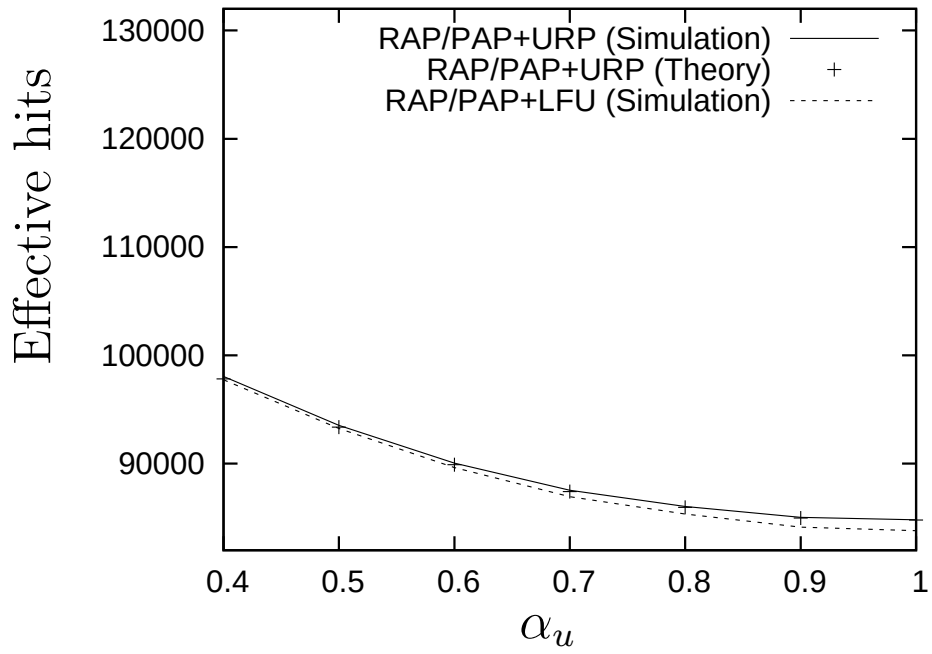


(a) Number of Effective Hits

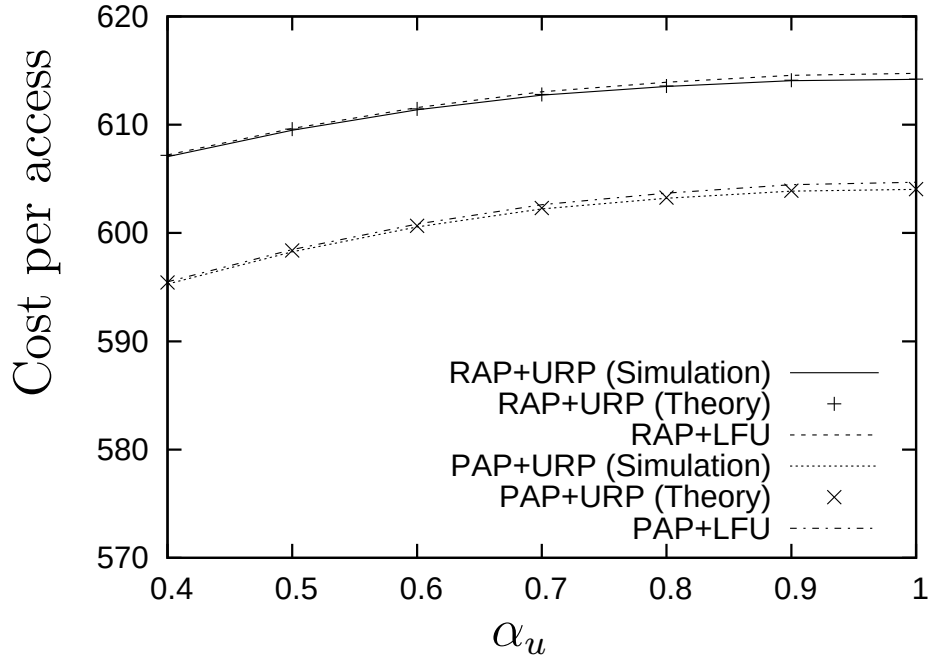


(b) Communication Cost

Figure 4.8: Performance of URP and LFU for different Zipf ratios ($\lambda = 0.50$)

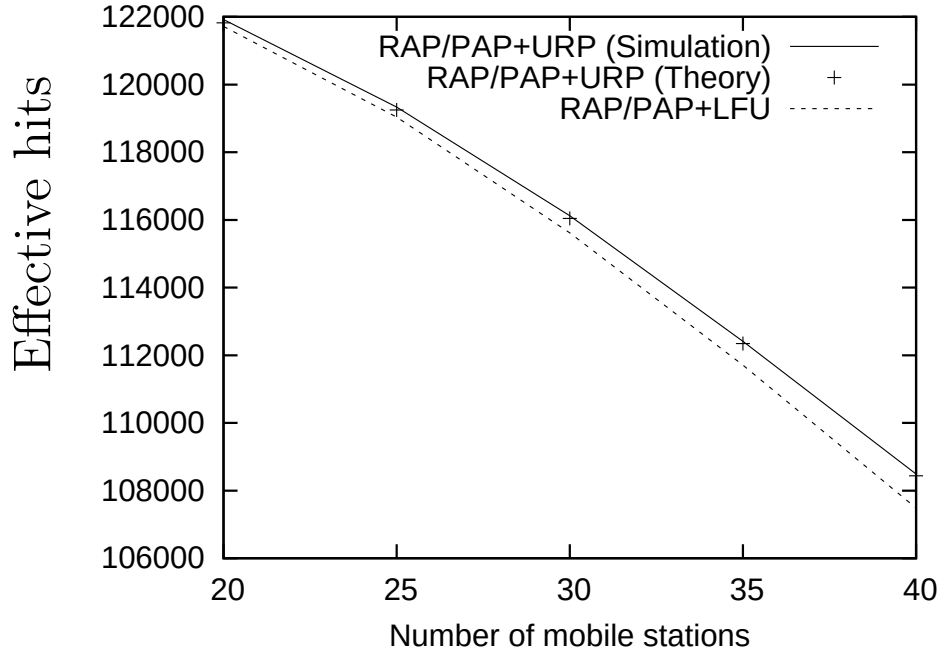


(a) Number of Effective Hits

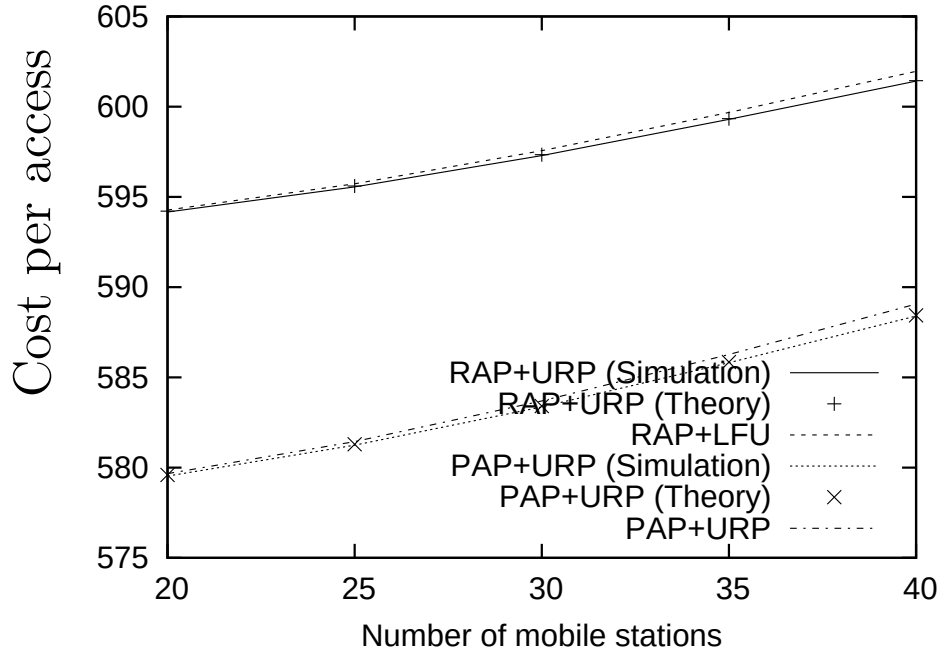


(b) Communication Cost

Figure 4.9: Performance of URP and LFU for different Zipf ratios ($\lambda = 0.60$)

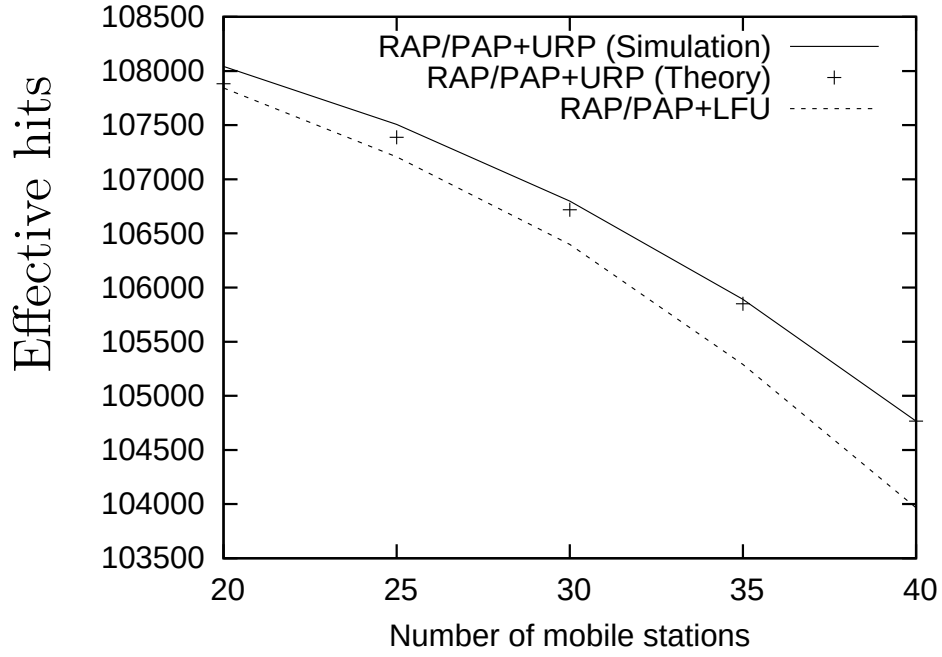


(a) Number of Effective Hits

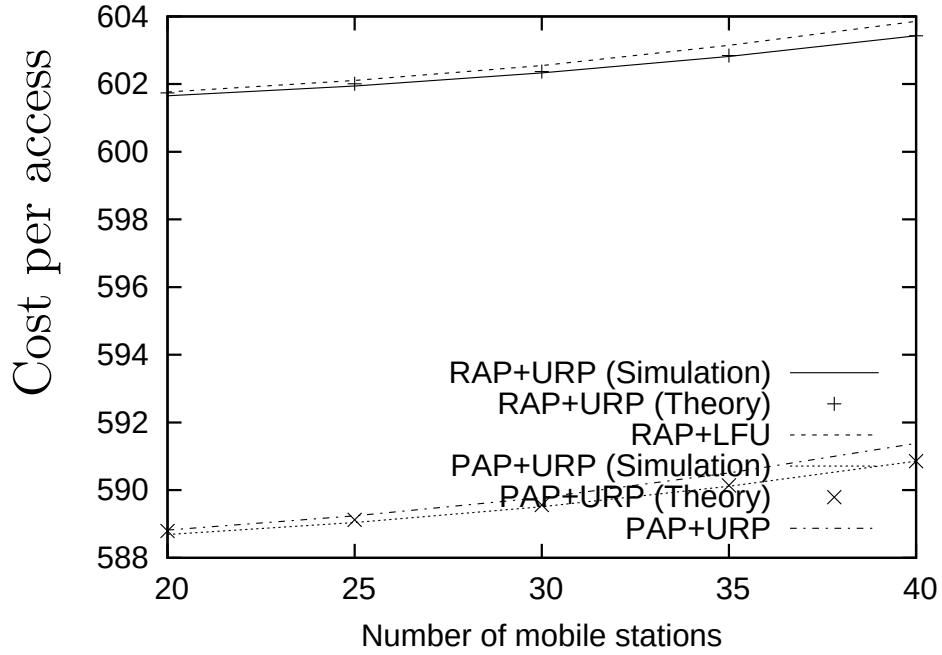


(b) Communication Cost

Figure 4.10: Performance of URP and LFU different number of mobile stations ($\lambda = 0.40$)



(a) Number of Effective Hits



(b) Communication Cost

Figure 4.11: Performance of URP and LFU different number of mobile stations ($\lambda = 0.50$)

4.4.5 Impact of Message Size

Finally, we investigate the impact of different message sizes, i.e., C_{msg}. We consult two cases when: (1) $K = 50$ and $N = 500$ and (2) $K = 100$ and $N = 400$. For both the cases, values of λ, α_a and α_u are set to 0.40, 0.01 and 0.60. The results are presented in Fig. 4.12. From both the results, it is evident that cost per access for URP increases at a faster rate than PAP as the ratio of C_{msg} to C_{obj} increases.

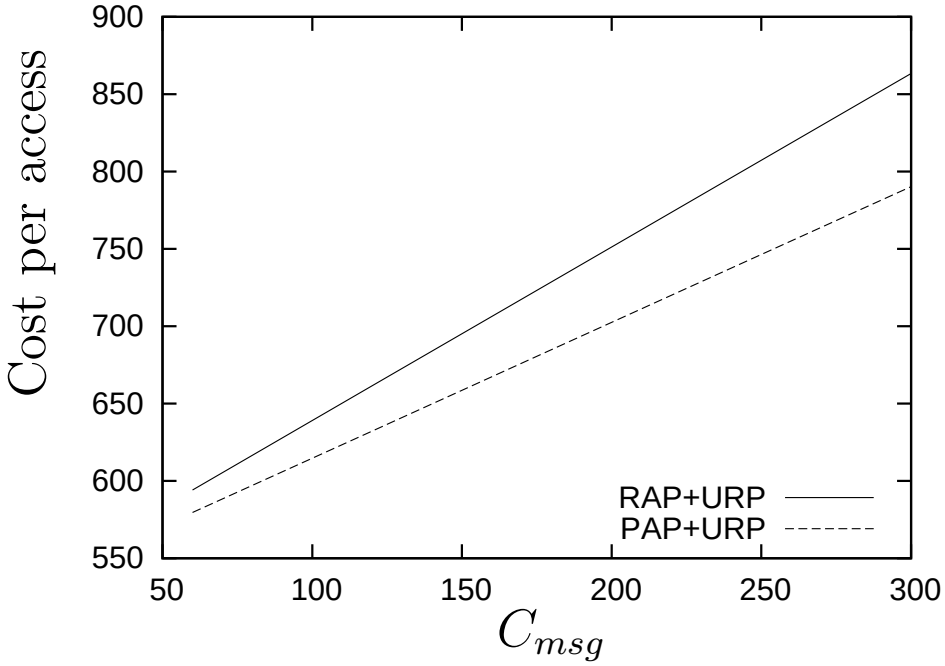
4.4.6 General Observations

We have the following general observations while preparing and running the simulator, and evaluating the results:

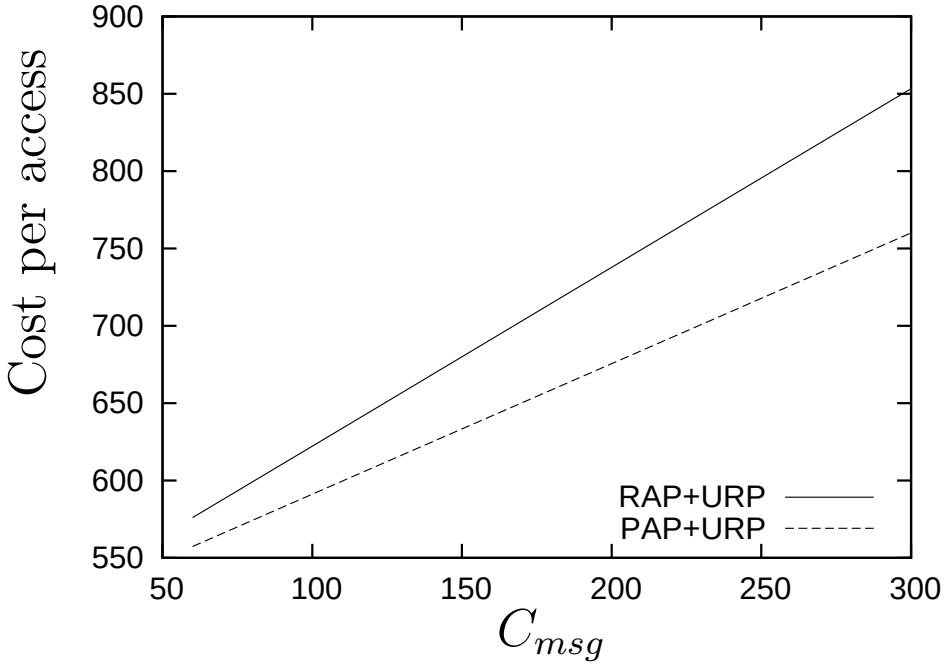
• Cases where cache suffer from a higher number of misses, or cases where there are fewer to no misses, the temporal objects introduced in between two consecutive cache replacements result in fewer extra cache hits. Thus, in those cases, the difference between theory and simulation subsides.

• Cases where cache suffers from a higher number of misses, the benefit of URP becomes more visible. As the system approaches to a system with readonly objects (i.e., objects with no updates), both LFU and URP approaches same (and optimal) performance.

• It is difficult to synthetically generate access and update traces which satisfy all the parameters and capture the worst case requirements. Hence, most of the traces are generated with high update rates and update Zipf ratios to make them behave more closer to the worst case scenarios. In all the simulations, the theoretical probability of hits are computed at each replacement only. Thus, simulation results are in general slightly better than the theoretical ones.



(a) Case 1



(b) Case 2

Figure 4.12: Cost of URP and LFU for different message sizes

Chapter 5

Client Injected Updates

In this chapter, we discuss the access and replacement for applications where the updates are injected from the clients only. The organization of this chapter is similar to that's of the previous chapter. We start with a discussion on update and access processes. Next, we analytically prove that if all the updates are injected only from the clients, a cache scheme, with our proposed policies is optimal in terms of effective hits and bandwidth consumption. In the final section of this chapter, we close our discussion by presenting the simulation results.

5.1 The Update Process

When updates are injected by the server only, the access mechanism focuses only on the consistency of data access. However, when updates are injected from the clients, the clients also need to take the responsibility of forwarding the update. In the later case, for both PAP and RAP, whenever a client initiates an update, the updated object is forwarded to the server. At the same time, both policies may store the updated object in the local cache in accordance with the replacement policy (described later). In PAP, as soon as an update from a client is received, the server is proactive in notifying all other clients hosting the same object about availability of the newer version. On the other hand, in RAP, a client reactively queries with the server about injected updates to a cached object from other clients, but only when the object is accessed.

5.2 The Update-oriented Replacement Policy (URP)

In this subsection, we present Update-oriented Replacement Policy (URP) for cache system with updates injected only from the clients. Like the policies in the previous chapter, here also both update and access frequencies are gathered through the access policies. We continue to use the tems *gain factor (GF)* to maintain consistency in discussion. However, we modify the definition of GF for the object O_i at client j up to time t as follows:

$$GF_i^j(t) = \frac{(\mu_i^j(t) + \lambda_i^j(t))\mu_i^j(t)}{\mu_i^j(t) + \lambda_i^{\mathbb{A}}(t)} \tag{5.1}$$

Similarly, long term GF is defined as,

$$GF_i^j = \lim_{t\to\infty} GF_i^j(t) = \lim_{t\to\infty} \frac{(\mu_i^j(t) + \lambda_i^j(t))\mu_i^j(t)}{\mu_i^j(t) + \lambda_i^{\mathbb{A}}(t)} = \frac{(\mu_i^j + \lambda_i^j)\mu_i^j}{\mu_i^j + \lambda_i^{\mathbb{A}}} \tag{5.2}$$

The algorithmic form of GF computation (gf_ci) is shown in Algorithm 3. In both PAP and RAP, whenever the primitive $find\_replaced$ is invoked gf_ci is considered to be the argument to assist in the computation of GF of objects.

Algorithm 3: *gf_ci* - GF Computation in algorithmic form

input : i is the objects identifier
input : j is the client identifier
output: Gain factor for requested object i at client j
return $\frac{(\mu_i^j+\lambda_i^j)\mu_i^j}{\mu_i^j+\lambda_i^{\mathbb{A}}}$;

5.3 Quantitative Analysis

In this section, we analyze a cache scheme employing URP for replacement and RAP/PAP for data access. We assume that all data update events are injected by the clients. Based on the behavior of URP policy, we conclude the following theorem.

Theorem 3 *URP maximizes the probability of guaranteed effective hits at each replacement for both RAP and PAP access policies when all updates are injected by the clients only.*

Proof: To prove the theorem, we follow the similar approach of proving Theorem. 1. We show that the probability of guaranteed effective hits after t-th replacement obtained by replacing $O_{URP}(t)$ with $O_a(t)$ is larger than or equal to the same probability obtained by replacing $O_r(t)$ with $O_a(t)$.

As event of access is a Poison process, the probability of accessing object O_i, $1 \leqslant i \leqslant N$, at client j, is

$$p_{a,i}^j = \frac{\mu_i^j}{\sum_{i=1}^N \mu_{i,j}} = \frac{\mu_i^j}{\mu^j} \tag{5.3}$$

Since event of update is also a Poison process, the probability of accessing or updating object O_i locally before being updated from any other client is[1],

$$\overline{p_{u,i}^j} = \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \sum_j \lambda_i^j} = \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \lambda_i^{\mathbb{A}}} \tag{5.4}$$

Note that $\overline{p_{u,i}^j}$ also represents the probability that O_i is locally accessed or updated before the object become invalid due to update from any other clients. Therefore, irrespective

[1]The detail derivation of the equation is discussed in Appendix E

of replacement policy in place, from (3.3), (5.3), and (5.4), we have,

$$
\begin{aligned}
P_{RAP}(t) &= P_{PAP}(t) \\
&= \sum_{\forall i | O_i \in C'(t)} p_{a,i}^j \overline{p_{u,i}^j} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} p_{a,i}^j \overline{p_{u,i}^j} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} \frac{\mu_i^j}{\mu^j} \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \lambda_i^{\mathbb{A}}} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\} \setminus \{O_r(t)\}} \frac{(\mu_i^j + \lambda_i^j)\mu_i^j}{(\mu_i^j + \lambda_i^{\mathbb{A}})\mu^j} \\
&= \sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu_i^j + \lambda_i^j)\mu_i^j}{(\mu_i^j + \lambda_i^{\mathbb{A}})\mu^j} - \frac{(\mu_r^j + \lambda_r^j)\mu_r^j}{(\mu_r^j + \lambda_r^{\mathbb{A}})\mu^j} \\
&= \frac{1}{\mu^j} \left[\sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \frac{(\mu_i^j + \lambda_i^j)\mu_i^j}{\mu_i^j + \lambda_i^{\mathbb{A}}} - \frac{(\mu_r^j + \lambda_r^j)\mu_r^j}{\mu_r^j + \lambda_r^{\mathbb{A}}} \right] \\
&= \frac{1}{\mu^j} \left[\sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} GF_i^j - GF_r^j \right]
\end{aligned}
\tag{5.5}
$$

When URP is utilized then, following equality can be driven in similar way:

$$
P_{RAP+URP}(t) = P_{PAP+URP}(t) = \frac{1}{\mu^j} \left[\sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} GF_i^j - GF_{URP}^j \right] \tag{5.6}
$$

Following the working strategy of the URP policy, when all updates are injected only from the clients, we can formulate,

$$
GF_{URP}^j \equiv \min_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \left(GF_i^j \right) \tag{5.7}
$$

Using (eq:11c) and (eq:12c), we conclude the following:

$$
\begin{aligned}
P_{RAP+URP}(t) &= P_{PAP+URP}(t) \\
&= \frac{1}{\mu^j} \left[\sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} GF_i^j - \min_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} \left(GF_i^j \right) \right] \\
&\geqslant \frac{1}{\mu^j} \left[\sum_{\forall i | O_i \in C(t) \cup \{O_a(t)\}} GF_i^j - GF_r^j \right] \\
&= P_{RAP}(t) = P_{PAP}(t)
\end{aligned}
\tag{5.8}
$$

From the above equality, it is evident that at each replacement, the probability of effective hits is the highest when URP is used by comparing with the same metric when another replacement policy is employed. This way, URP maximizes the probability of guaranteed effective hits at each replacement for both RAP and PAP, when all updates are injected only by the clients.■

With the above proof available, we can deduce the following corollary.

Corollary 3 *When all the data updates are injected by the clients, URP minimizes the expected cost of data access at each replacement for both RAP and PAP.*

We prove the corollary in Appendix B. Based on the properties of our proposed policies, proven above, we assert the following Theorem:

Theorem 4 *In a cache system where all updates are injected from the clients, in the long run, URP gives optimal guaranteed effective cache hits for both RAP and PAP.*

Proof of the theorem is presented in Appendix C. Beside the above, we can also add that,

Corollary 4 *In a cache system where all updates are injected from the clients, in the long run, URP guarantees optimal cost for both RAP and PAP.*

Proof of the corollary is presented in Appendix D.

5.4 Performance Evaluation

We present different results available from our extensive simulations in this section. We use the same simulation traces, we used for the simulations presented in the last chapter, except one difference. In simulations for this chapter, all update events are distributed among the clients, where as for previous chapter, all update events are considered take place at the server. Though the observations from the simulation results of this and the previous chapter are comparable, a detail discussion is presented for the sake of completeness.

5.4.1 Impact of Objects Population

In this subsection, we discus our study on the effect of object population size (N). In Fig. 5.1 and 5.2, the performance of RAP/PAP+URP is compared with RAP/PAP+LFU for different object population sizes. We consider the values for the parameters α_a, α_u and K in these simulations to be 0.20, 0.60 and 20, respectively. We observe following behaviors from these results.

• If the object population size increases as compared to the size of the cache buffer, a smaller portion of the object population can be cached. Thus, irrespective of the replacement policy, the probability of effective cache hit reduces. However, in all cases, URP shows better results.

• PAP+URP policy demonstrates the best performance among the four combinations. However, for smaller object population PAP+LFU performs very closely. When N is smaller, in both, a cache can accommodate all the higher utility objects irrespective of the replacement. Similar argument is valid for RAP+URP and RAP+LFU policy.

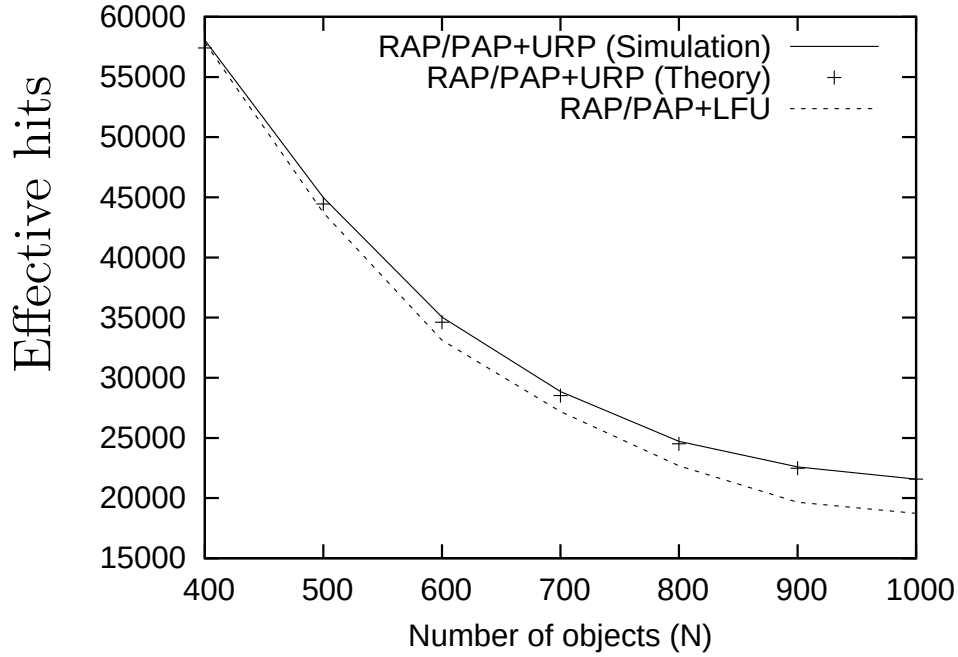
• The gain of using URP becomes further evident with larger population sizes.

• Cost-wise, both PAP policy combinations perform better than RAP combinations, and in each group URP demonstrates superior performance.

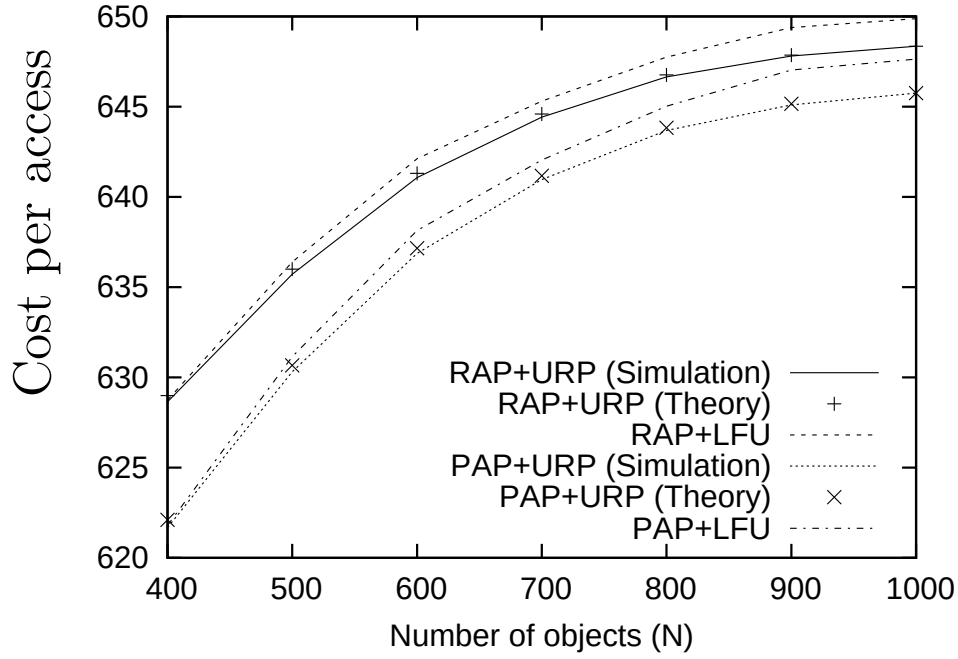
5.4.2 Impact of Cache Size

Performance characteristics for different cache sizes are shown in Fig. 5.3 and 5.4. For these simulations, the values for the parameters α_a, α_u and N are chosen to be 0.01, 0.60 and 400, respectively. From the figures, we deduce following arguments:

• Increasing cache size has opposite effect of increasing object popularity – as cache size increases, number of hits increases. Larger cache allows more objects to be in the cache, at a given time, resulting in higher number of hits for all cache schemes.

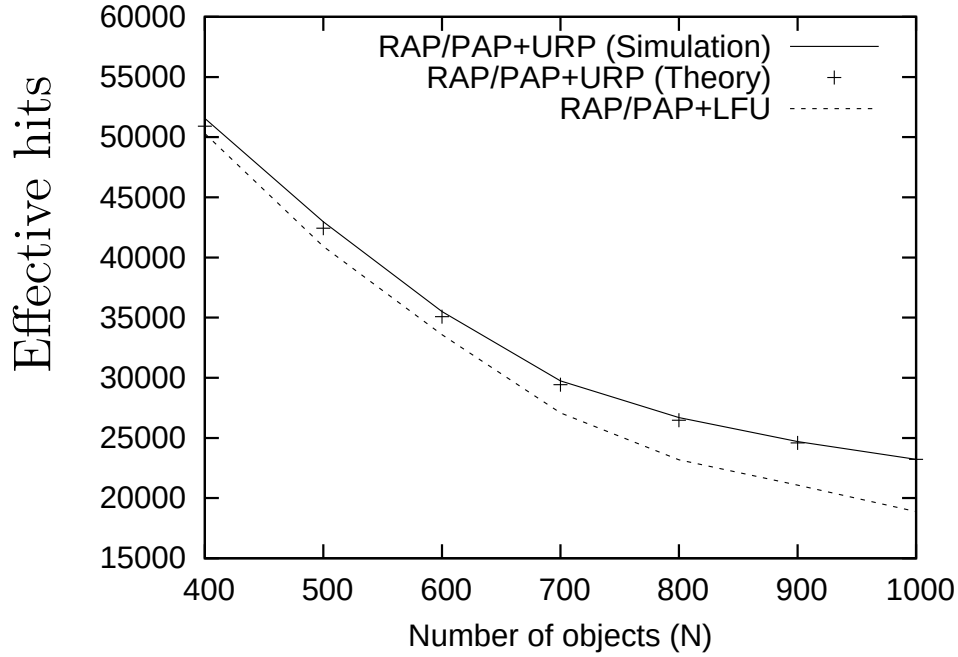


(a) Number of Effective Hits

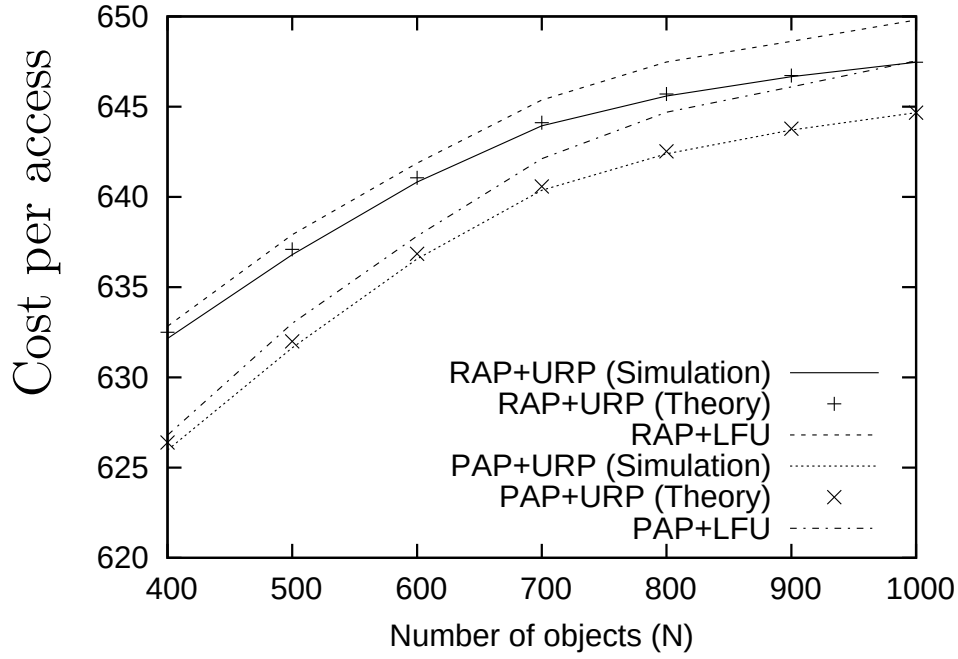


(b) Communication Cost

Figure 5.1: Performance of URP and LFU for different object population ($\lambda = 0.40$)



(a) Number of Effective Hits



(b) Communication Cost

Figure 5.2: Performance of URP and LFU for different object population ($\lambda = 0.60$)

• At the same time, URP demonstrate superior performance in all cases.

• With larger cache, the performance difference between URP and LFU diminishes. Though their replacement criterion are different and preference ranks (GF for URP and access frequency for LFU) for objects are also different, large cache allow the policies to store wider preferences of objects and thus both the policies host a significant amount of common contents.

• With increasing cache size, increasing number of hits helps in reducing the cost per access for all combinations of cache policies.

• With increment of cache size, each additional extra cache buffer results in fewer additional effective hits. Hence, a designer must find a trade off between the increment of cache performance and the cost of adding extra cache buffer space.

5.4.3 Impact of Objects with No updates

Number of effective hits and cost per access for URP and LFU policy when no update to any object take place, i.e., λ_i^j for all i, j is 0 are shown in Fig. 5.5, 5.6 and 5.7. In other word, these objects are for read only. To gather these results, in all the simulations, an object population size of 500 is considered. With no update events, GF_i^j is determined by μ_i^j (as can be determined from (5.2)) and as a result, while making a replacement (as well as, eviction) decision, both URP and LFU choose the same object. Thus both the policies result in exactly the same performance.

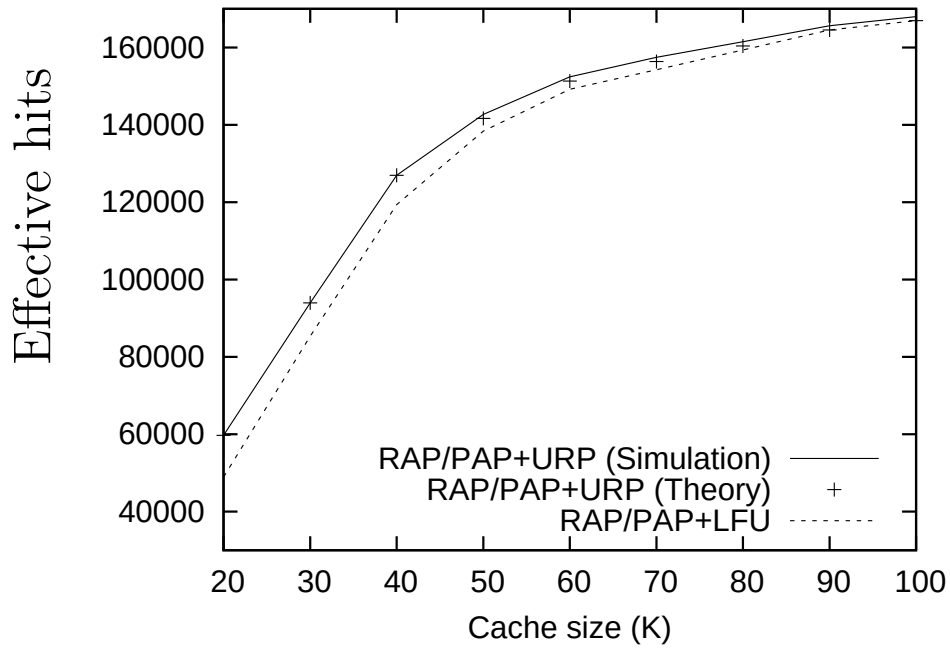
These results are in perfect match with the results for the same scenario presented in Chapter 4.

5.4.4 Impact of Zipf Ratio

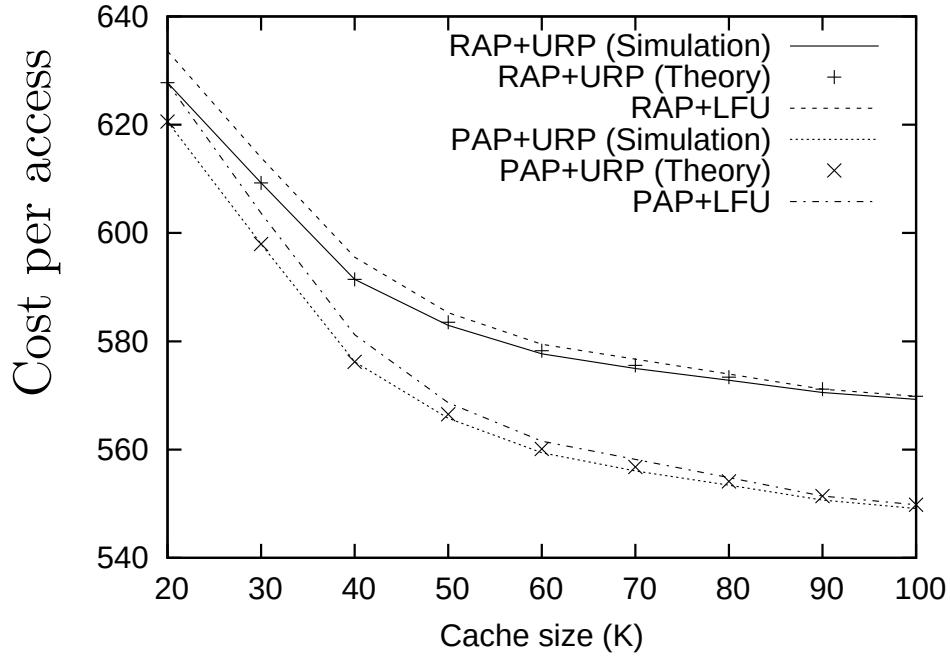
The effect of update Zipf ratio on the performance of both URP and LFU policies is shown and compared in Fig. 5.8 and 5.9. The parameters α_a, K and N are set to 0.01, 50 and 500, respectively, for these simulations. The results establish that:

• In these simulations, while computing Zipf ratios, we have considered that both access and update ranks of object O_i are i. As a result, more frequent updates to more frequently accessed objects results in fewer overall hits. Note that, if rank for access and update Zipf ratio for all objects are arbitrary, the performance of the system may approach to a system with no update at all (see subsection 5.4.3), depending on the values for Zipf ratios.

• As usual, cache schemes with URP demonstrate superior results.

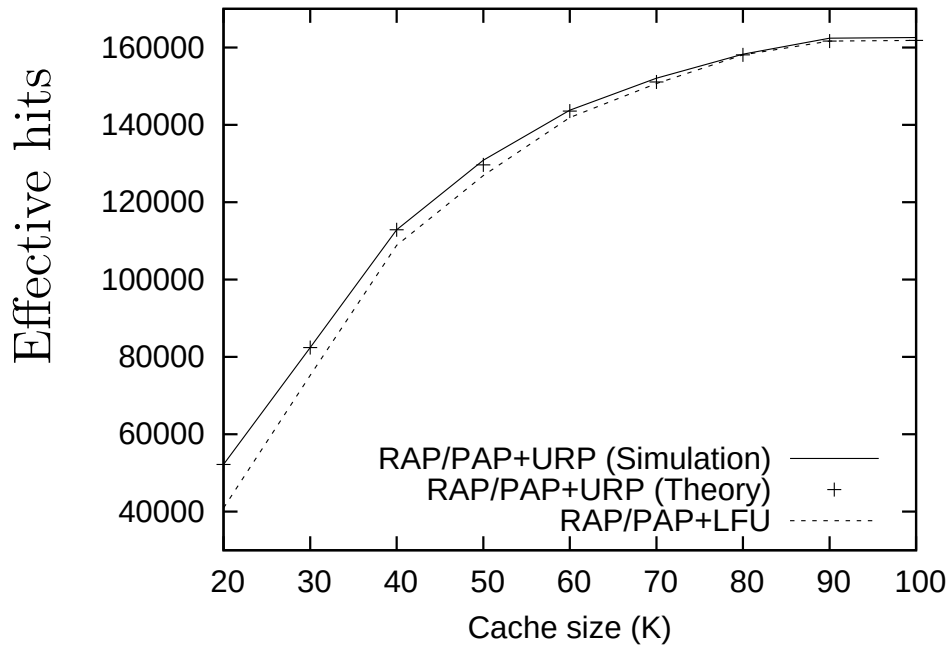


(a) Number of Effective Hits

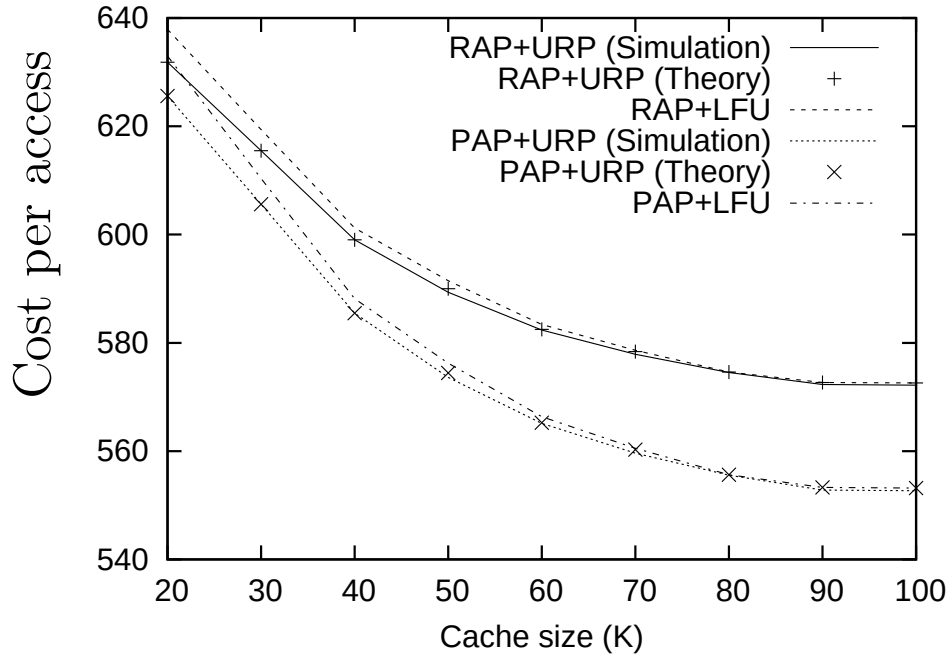


(b) Communication Cost

Figure 5.3: Performance of OUR and LFU for different cache sizes ($\lambda = 0.50$)

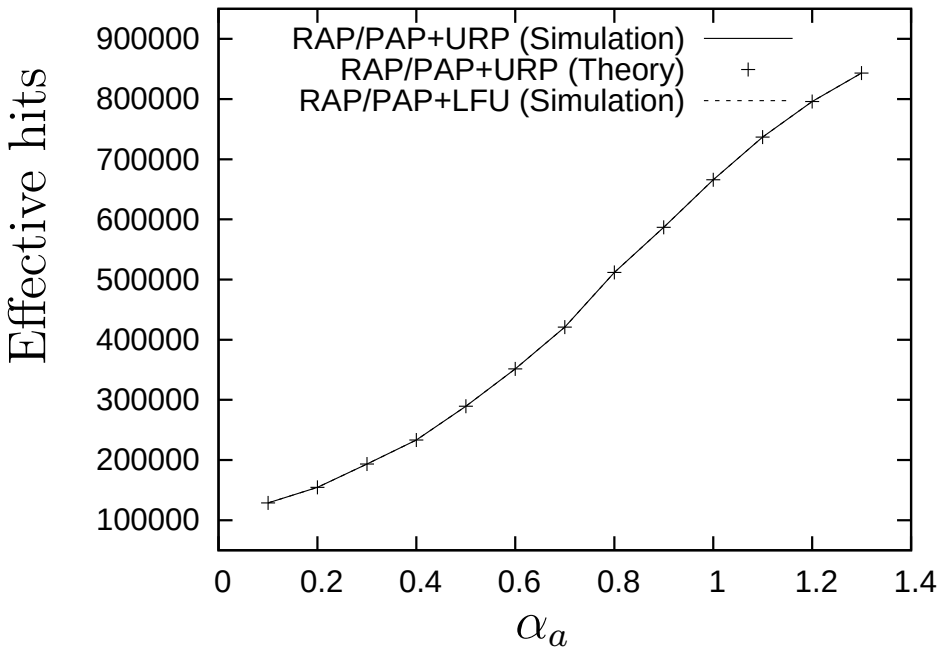


(a) Number of Effective Hits

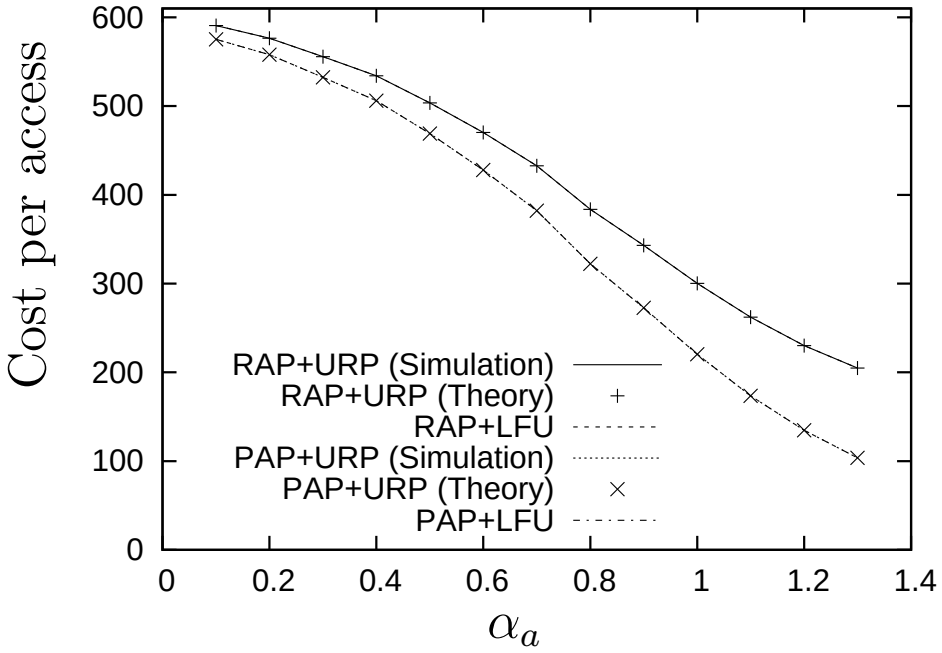


(b) Communication Cost

Figure 5.4: Performance of OUR and LFU for different cache sizes ($\lambda = 0.60$)

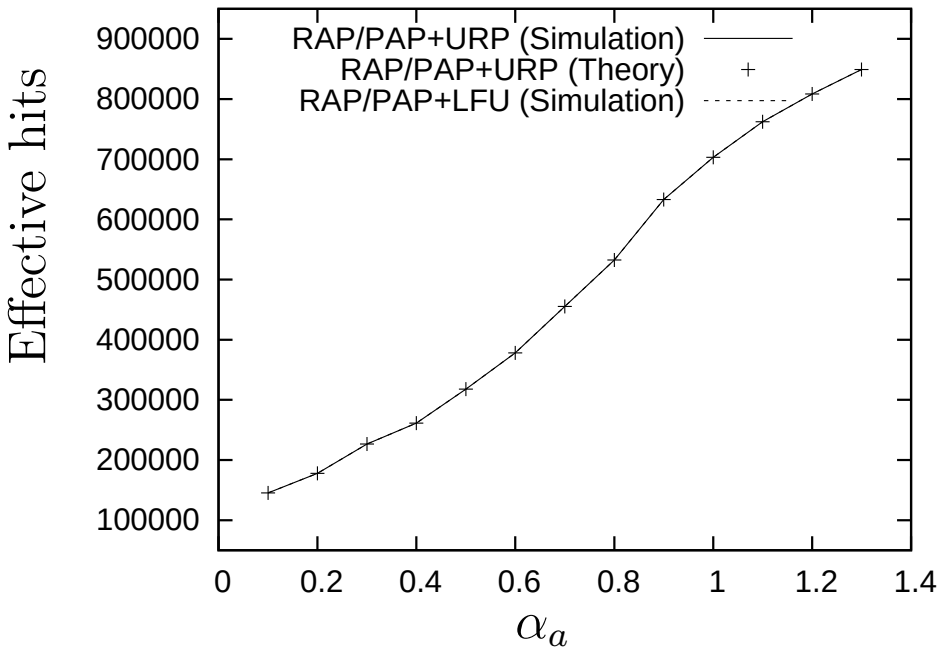


(a) Number of Effective Hits

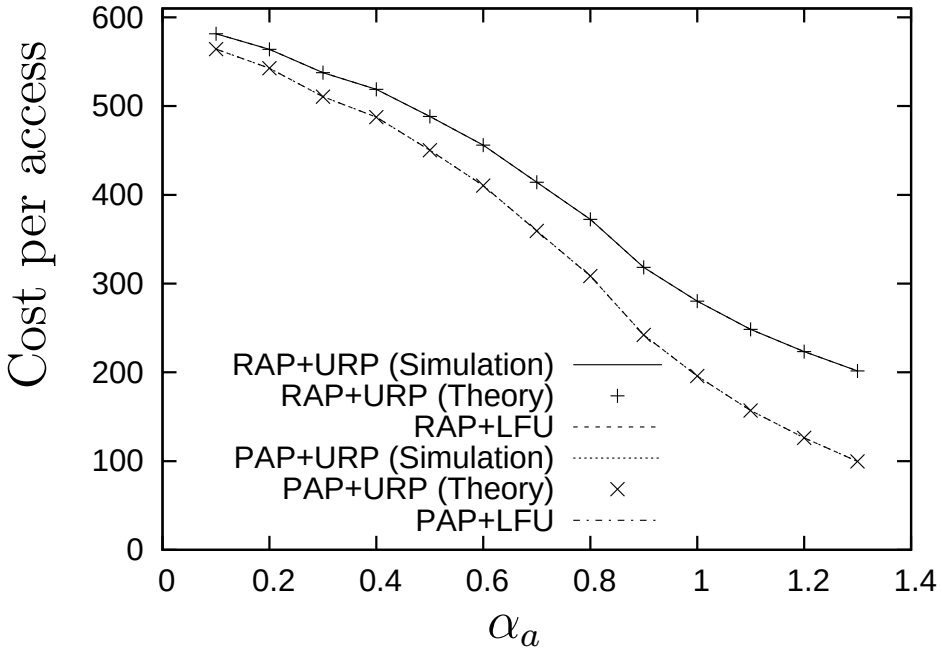


(b) Communication Cost

Figure 5.5: Performance of URP and LFU for objects with no update ($K = 50$)

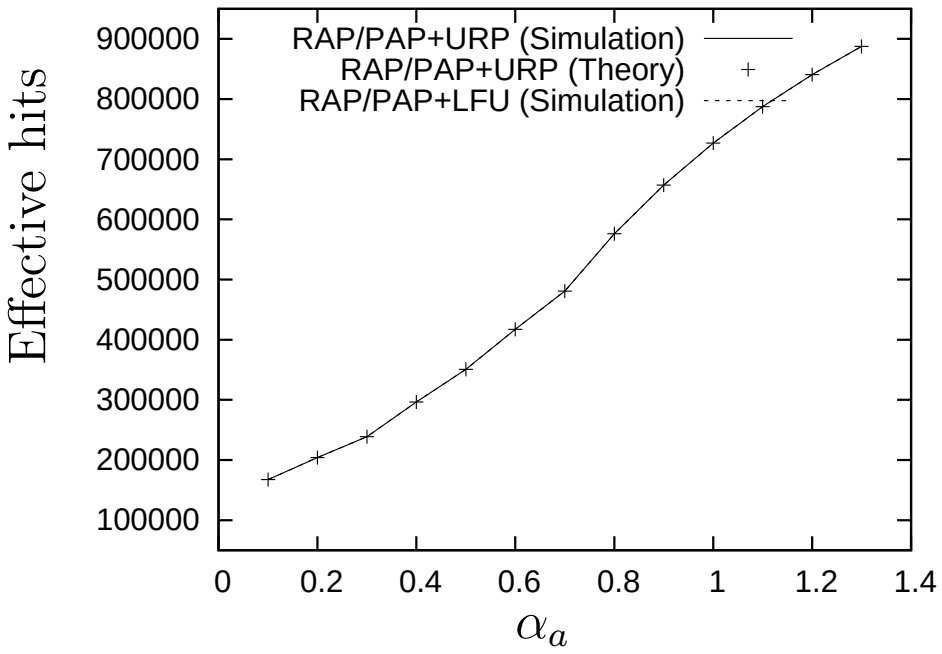


(a) Number of Effective Hits

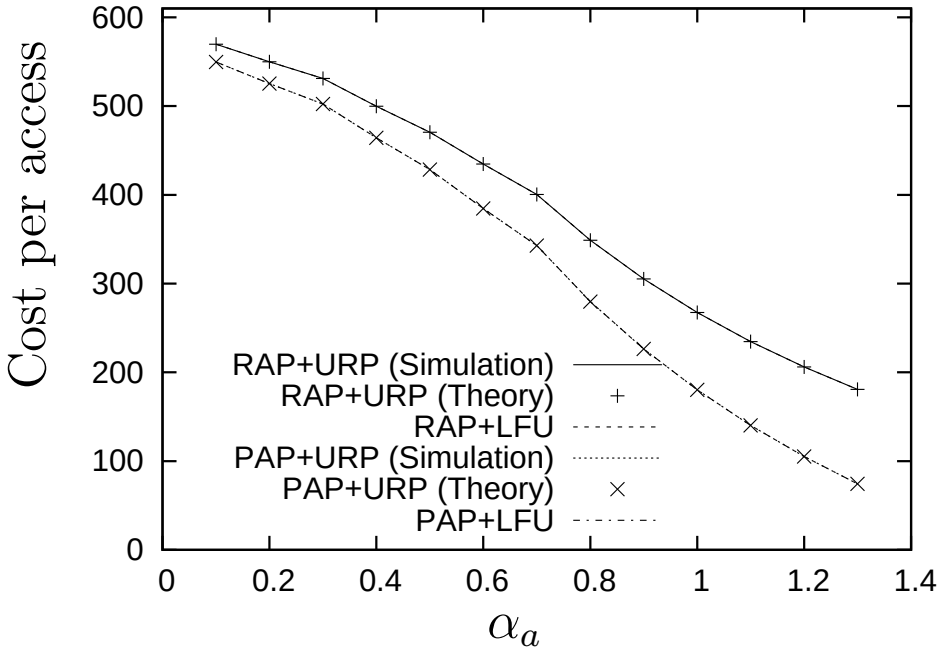


(b) Communication Cost

Figure 5.6: Performance of URP and LFU for objects with no update ($K = 60$)

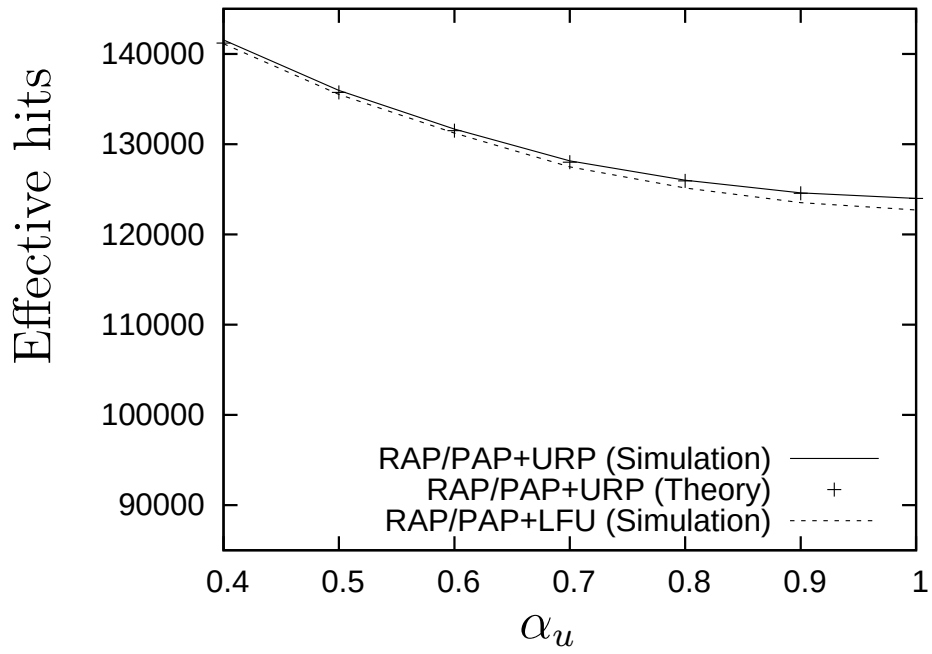


(a) Number of Effective Hits

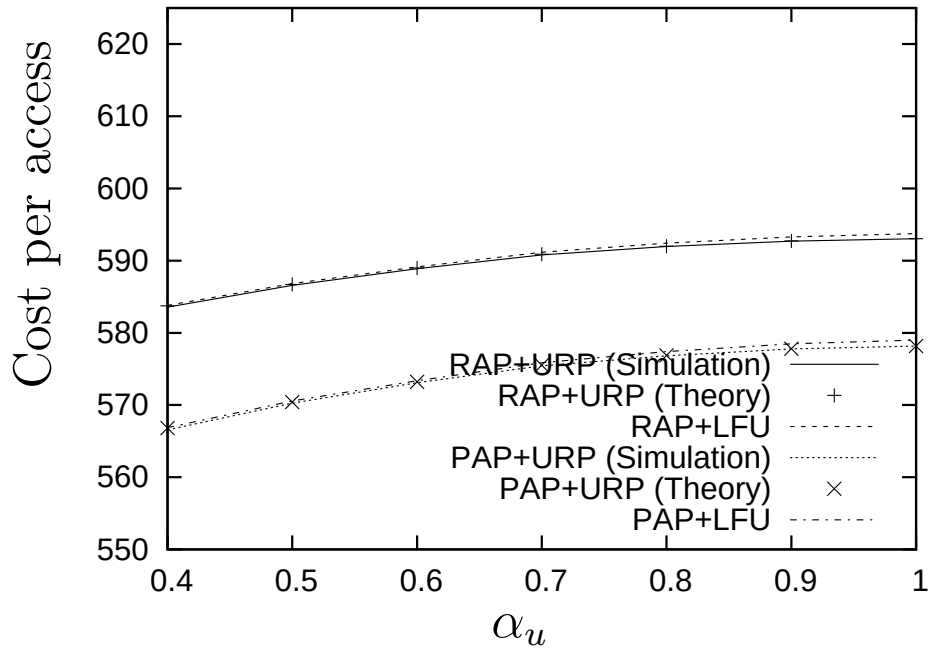


(b) Communication Cost

Figure 5.7: Performance of URP and LFU for objects with no update ($K = 70$)

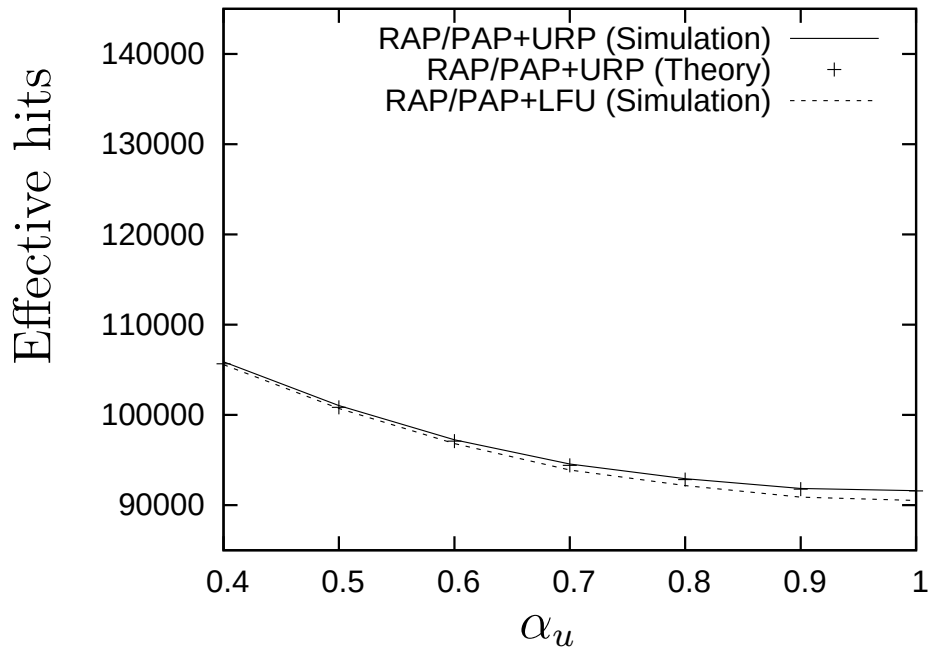


(a) Number of Effective Hits

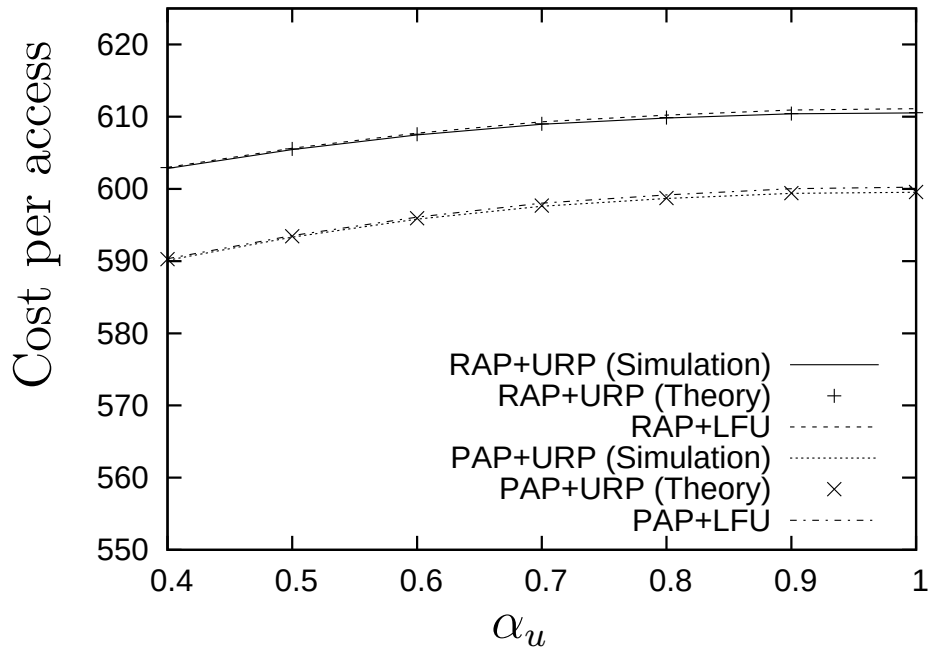


(b) Communication Cost

Figure 5.8: Performance of OUR and LFU for different Zipf ratios ($\lambda = 0.50$)



(a) Number of Effective Hits



(b) Communication Cost

Figure 5.9: Performance of OUR and LFU for different Zipf ratios ($\lambda = 0.60$)

5.4.5 Impact of Number of Mobile Stations

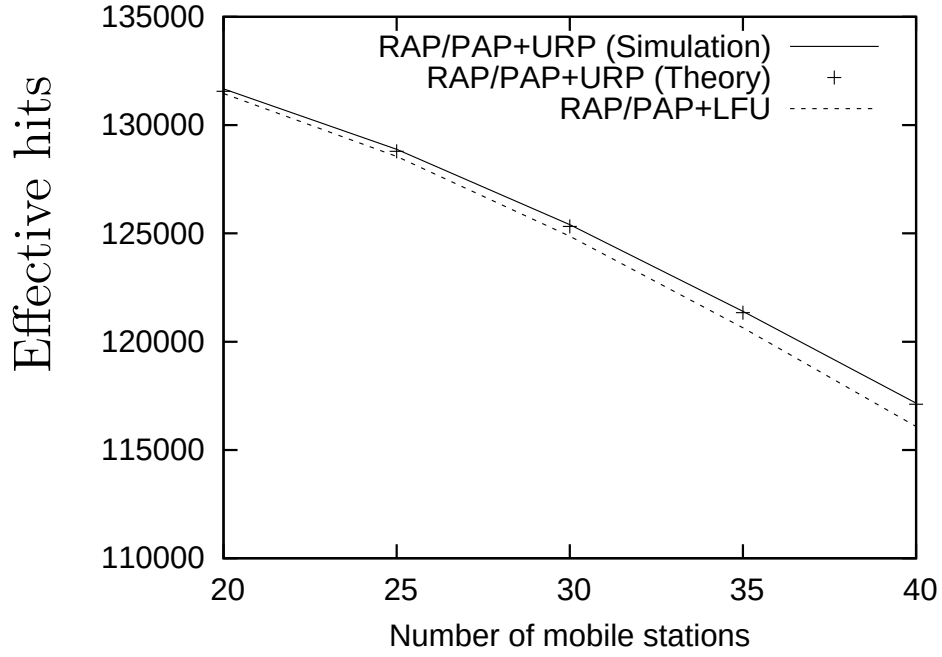
In (5.2), given a system wide update rate, $\lambda_i^{\mathbb{A}}$ is a global metric and the number of mobile stations has no effect on it. On the other hand, for a given μ_i and λ_i, μ_i^j and λ_i^j are affected by the number of mobile stations. With increasing number of mobile stations, the variance of GFs of different objects becomes smaller and λ_i turns out to be the main determining factor. Fig 5.10 and 5.11 presents the simulation results for different number of mobile stations. The simulation parameters α_a, α_u, K, and N are assigned values 0.01, 0.06, 50, and 500, respectively. As shown in the figures, with more MSes the number of effective hits decreases and the cost per access increases.

5.4.6 Other Observations

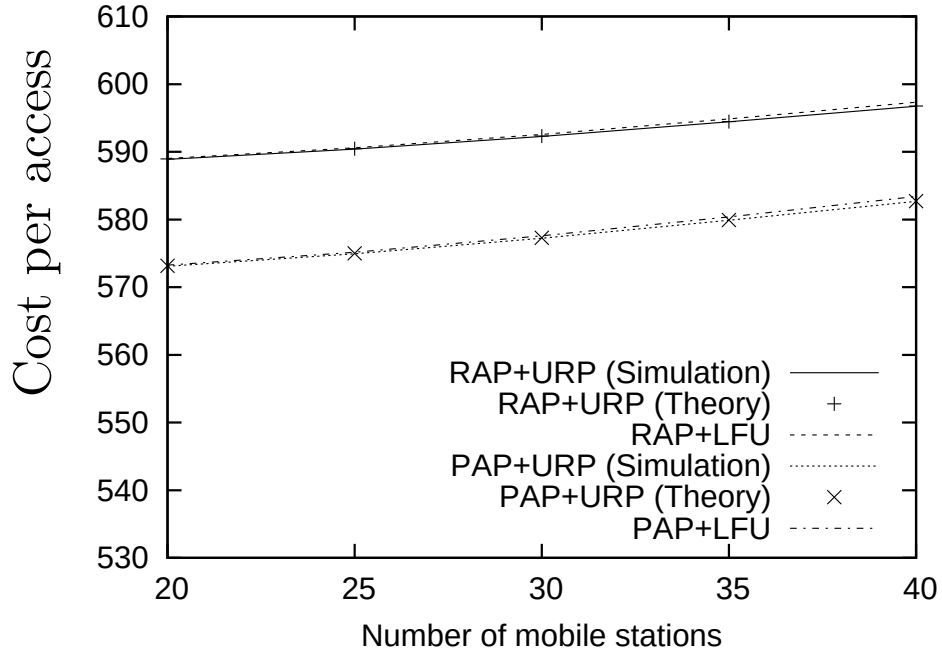
All observations presented in Subsection 4.4.6 are equally valid for cache systems where all updates are injected from the clients. However, we have following additional observations:

• The numbers of effective hits presented in this chapter are higher (with the exception of readonly object scenario) than the numbers for the corresponding scenarios presented in the last chapter. When updates are injected from the server only, a update render all the copies of the updated object obsolete. In contrast, update to higher utility (GF) object at a client also inject the updated object into the client cache, increasing the chance of local cache hit.

• For the same reason, discussed above, the expected cost for data access is also lower for the cache systems discussed in this chapter.

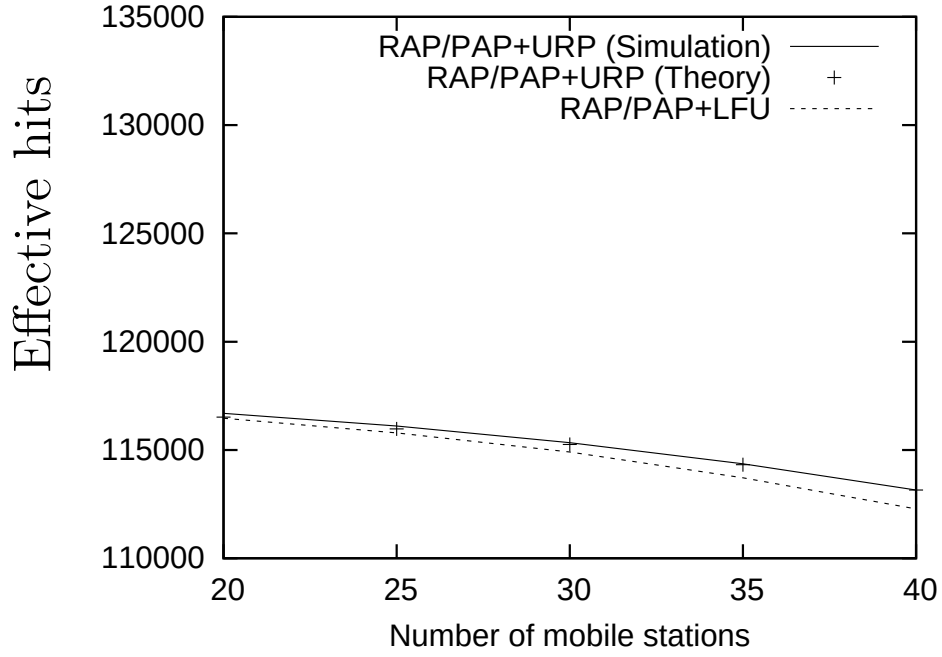


(a) Number of Effective Hits

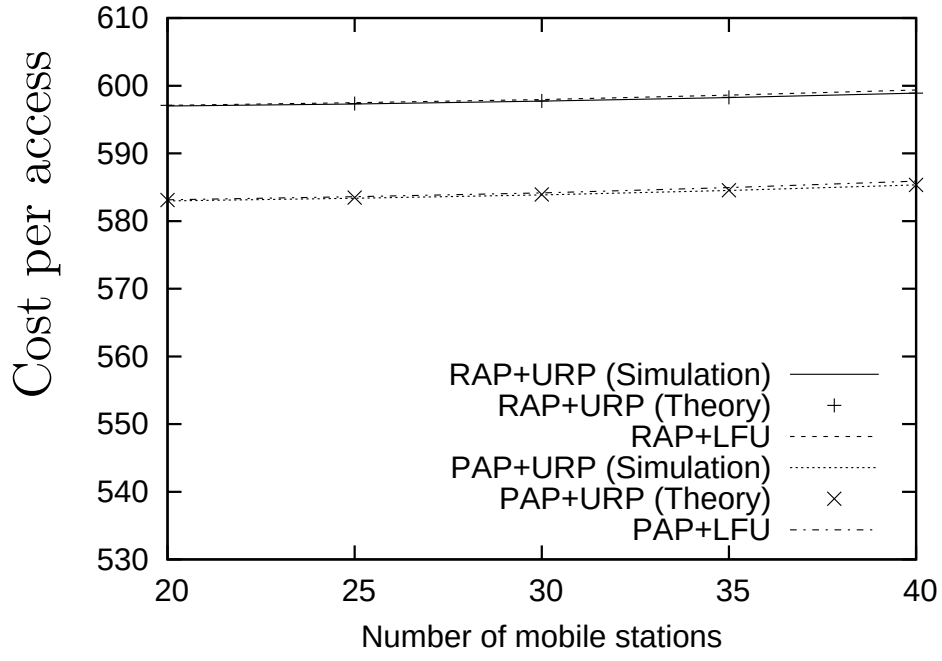


(b) Communication Cost

Figure 5.10: Performance of OUR and LFU different number of mobile stations ($\lambda = 0.40$)



(a) Number of Effective Hits



(b) Communication Cost

Figure 5.11: Performance of OUR and LFU different number of mobile stations ($\lambda = 0.50$)

Chapter 6

Conclusion

In this chapter, we summarize our contributions and present potential future works.

6.1 Contribution

In this thesis, we have proposed an optimal cache replacement policy, named *Update-oriented Replacement Policy* (URP) for wireless data access applications. To maintain strong consistency among copies of objects and facilitate working environment for URP, we have also proposed two enhanced cache access policies – *Proactive Access Policy* (PAP), and *Reactive Access Policy* (RAP). We have considered the impact of data updates injected in the system. We have emphasized two kinds of update events - firstly, all updates are injected by the server and secondly, all all updates are injected by the clients. In Appendix A, we have extended the solutions to systems where updates are injected from both the server and the clients.

The goal of the proposed policies is to make efficient use of the network bandwidth in wireless environment, by increasing effective cache hits. We have proved that if PAP or RAP is combined with URP, the cache system guarantees optimal number of effective cache hits and optimal cost (in terms of network bandwidth) per data object access. Due to our comprehensive system model, the proposed policies are equally applicable to existing 2G and 3G, as well as upcoming Long Term Evolution (LTE), LTE Advanced and WiMAX wireless data access networks.

6.2 Future Works

Our research in this thesis has been bounded by the mobile wireless data access networks. We are planning to extend our focus to the optimal caching schemes for other types of wireless network infrastructures, such as networks supporting limited broadcast, such as WLANs. We are particularly interested in - (1) multi-hop wireless networks and (2) networks combining mobile and geographically static wireless network (such as network formed by smartphones with 3G and WLAN capability).

Recently, we are also interested in investigating the problem of cooperative caching in wireless networks. In this utopian system, the participants in the entire system collaborate with each other to satisfy each others needs. Clients can be opportunistic in fetching objects that are requested by other clients. At the same time, an idle client can assist other neighboring busy clients to cache objects for future use.

Finally, We are actively looking forward to finding solution for data dissemination in wireless networks using efficient data caches.

Bibliography

[1] Mursalin Akon, Towhidul Islam, Xuemin Shen, and Ajit Singh. SPACE: A lightweight collaborative caching for clusters. *Peer-to-Peer Networking and Applications*, 3(2):80–96, 2010.

[2] AT&T - News Room. AT&T launches pilot Wi-Fi project in Times Square, 2010. `http://www.att.com/gen/press-room?pid=4800&cdvn=news& newsarticleid=30838`.

[3] AT&T - News Room. AT&T Wi-Fi handles more than 85 million total connections in 2009, more than four times 2008, 2010.

[4] AT&T - News Room. AT&T Wi-Fi network usage soars to more than 53 million connections in the first quarter, 2010.

[5] Daniel Barbará and Tomasz Imieliński. Sleepers and workaholics: caching strategies in mobile environments. *ACM SIGMOD Record*, 23(2):1–12, 1994.

[6] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1-2):119–125, 1995.

[7] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM*, volume 1, pages 126–134, March 1999.

[8] Jun Cai and Kian-Lee Tan. Energy-efficient selective cache invalidation. *Wireless Networks*, 5(6):489–502, 1999.

[9] Guohong Cao. Proactive power-aware cache management for mobile computing systems. *IEEE Transaction on Computers*, 51(6):608–621, 2002.

[10] Guohong Cao. A scalable low-latency cache invalidation strategy for mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1251–1265, 2003.

[11] Boris Y. L. Chan, Antonio Si, and Hong Va Leong. Cache management for mobile databases: Design and evaluation. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 54–63, 1998.

[12] Hui Chen, Yang Xiao, and Xuemin Shen. Update-based cache access and replacement in wireless data access. *IEEE Trans. on Mobile Computing*, 5(12):1734–1748, 2006.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[14] Gerard Bosch I Creus and Petri Niska. System-level power management for mobile devices. In *International Conference on Computer and Information Technology*, pages 799–804, 2007.

[15] Dropbox, Inc. Dropbox, 2010.

[16] Facebook, Inc. facebook, 2010.

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[18] Cedric C. F. Fong, John C. S. Lui, and Man Hon Wong. Quantifying complexity and performance gains of distributed caching in a wireless mobile computing environment. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 104–113, 1997.

[19] Google Inc. Picasa, 2010.

[20] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[21] Qinglong Hu and Dik Lun Lee. Cache algorithms based on adaptive invalidation reports for mobile environments. *Cluster Computing*, 1(1):39–50, 1998.

[22] Jin Jing, Ahmed Elmagarmid, Abdelsalam Sumi Helal, and Rafael Alonso. Bit-Sequences: A new cache invalidation method in mobile environments. *Mobile Networks and Applications*, 2(2):115–127, 1997.

[23] A. Kahol, S. Khurana, S.K.S. Gupta, and P.K. Srimani. A strategy to manage cache consistency in a distributed mobile wireless environment. *IEEE Transaction on Parallel and Distributed Systems*, 12(7):686–700, 2001.

[24] Ajey Kumar, Anil K. Sarje, and Manoj Misra. Prioritised predicted region based cache replacement policy for location dependent data in mobile environment. *International Journal of Ad Hoc and Ubiquitous Computing*, 5(1):56–67, 2010.

[25] Yi-Bing Lin, Wei-Ru Lai, and Jen-Jee Chen. Effects of cache mechanism on wireless data access. *IEEE Transactions on Wireless Communications*, 2(6):1247–1258, 2003.

[26] Alok Madhukar, Tansel zyer, and Reda Alhajj. Dynamic cache invalidation scheme for wireless mobile environments. *Wireless Networks*, 15(6):727–740, 2009.

[27] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[28] Xuemin (Sherman) Shen Mursalin Akon, Mohammad Towhidul Islam and Ajit Singh. Hit optimal cache for wireless data access. In *IEEE Globecom*, 2010. (accepted to).

[29] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM SIGOPS Operating Systems Review*, 21(5):3–4, 1987.

[30] News Corp. Digital Media. Myspace, 2010.

[31] Gian Paolo Perrucci, Frank Fitzek, Giovanni Sasso, Wolfgang Kellerer, and Joerg Widmer. On the impact of 2G and 3G network usage for mobile phones' battery life. In *European Wireless 2009*, pages 255–259, 2009.

[32] Photobucket. Photobucket, 2010.

[33] Qzone QQ. Qzone, 2010.

[34] Lakshmish Ramaswamy and Ling Liu. An expiration age-based document placement scheme for cooperative Web caching. *IEEE Transactions on Knowledge and Data Engineering*, 16(5):585–600, 2004.

[35] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *ACM SIGMETRICS Performance Evaluation Review*, volume 18, pages 134–142, 1990.

[36] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[37] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, NJ, USA, 2002.

[38] The University of California, San Diego. Squid, 2010.

[39] Kian-Lee Tian, Jun Cai, and Beng Chin Ooi. An evaluation of cache invalidation strategies in wireless environments. *IEEE Transactions on Parallel and Distributed Systems*, 12(8):789–807, 2001.

[40] Twitter, Inc. twitter, 2010.

[41] ccache under GNU General Public License. ccache, 2010.

[42] Xian Wang and Pingzhi Fan. A strongly consistent cached data access algorithm for wireless data networks. *Wireless Networks*, 15(8):1013–1028, 2009.

[43] Wikimedia Foundation, Inc. Domain name system, 2010.

[44] Windows Live SkyDrive. Skydrive, 2010.

[45] Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Energy-efficient caching for wireless mobile computing. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 336–343, 1996.

[46] Richard S. L. Wu, Allan K. Y. Wong, and Tharam S. Dillon. E-MACSC: A novel dynamic cache tuning technique to reduce information retrieval roundtrip time over the Internet. *Computer Communications*, 29(8):1094–1109, 2006.

[47] Yahoo! Inc. flickr, 2010.

[48] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transaction on Knowledge and Data Engineering*, 11(4):563–576, 1999.

[49] Youku. Youku, 2010.

[50] YouTube, LLC. Youtube, 2010.

[51] Joe Chun-Hung Yuen, Edward Chan, Kam-Yiu Lam, and H. W. Leung. Cache invalidation scheme for mobile computing systems with real-time data. *ACM SIGMOD Record*, 29(4):34–39, 2000.

[52] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive Web Caching: Towards a new global caching architecture. In *3rd International WWW Caching Workshop*, pages 2169–2177, 1998.

[53] Yingwu Zhu and Yiming Hu. Exploiting client caches to build large Web caches. *The Journal of Supercomputing*, 39(2):149–175, 2007.

Appendices

Appendix A

Both Server and Clients Injected Updates

In this Appendix, we present our proposed replacement policy and access policies for cache systems allowing updates to data object from both the clients and the server.

A.1 The Update Process

The update process is simply a composition of the steps defined in previous chapters. When an object update is injected from the server, the steps described in Section 4.1 is followed. Whereas, if the update is injected from a client, the steps in Section 5.1 is followed.

A.2 The Update-oriented Replacement Policy (URP)

Gain factor (GF) the object O_i at client j up to time t for cache systems allowing updates from both the clients and server is computed as follows:

$$GF_i^j(t) = \frac{(\mu_i^j(t) + \lambda_i^j(t))\mu_i^j(t)}{\mu_i^j(t) + \lambda_i(t)} \tag{A.1}$$

Similarly, long term GF is defined as,

$$GF_i^j = \lim_{t\to\infty} GF_i^j(t) = \lim_{t\to\infty} \frac{(\mu_i^j(t) + \lambda_i^j(t))\mu_i^j(t)}{\mu_i^j(t) + \lambda_i(t)} = \frac{(\mu_i^j + \lambda_i^j)\mu_i^j}{\mu_i^j + \lambda_i} \tag{A.2}$$

The algorithmic form of GF computation (gf_csi) is shown in Algorithm 4. In an implementation of PAP and RAP, each call to $find\_replaced$ (as in Fig. 3.3 and 3.4) is augmented with gf_csi as an argument.

Algorithm 4: *gf_csi* - GF Computation in algorithmic form

input : i is the objects identifier
input : j is the client identifier
output: Gain factor for requested object i at client j
return $\frac{(\mu_i^j+\lambda_i^j)\mu_i^j}{\mu_i^j+\lambda_i}$;

A.3 Quantitative Analysis

In this section, we analyze PAP and RAP for data access, combined with URP for replacement for cache systems where updates are injected from both the clients and the server. We conclude the following theorem based on the working principle of the URP policy.

Theorem 5 *For cache systems where updates are injected from both the clients, as well as the server, the probability of guaranteed effective hits at each replacement is maximized for both RAP and PAP access mechanism when as replacement policy URP is in place.*

Proof: We prove the theorem by showing that no other replacement policy exists that can result in higher probability of guaranteed effective hits after t-th replacement than the case when URP is exercised.

At client j, at any access, the probability of the accessed object being a given object O_i, $1 \leqslant i \leqslant N$, is

$$p_{a,i}^j = \frac{\mu_i^j}{\sum_{i=1}^{N} \mu_{i,j}} = \frac{\mu_i^j}{\mu^j} \tag{A.3}$$

Note that access event is a Poison process. So, the probability of local access or update to O_i before it is updated by any other clients or the server is,

$$\overline{p_{u,i}^j} = \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \sum_j \lambda_i^j + \lambda_i^{\mathbb{S}}} = \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \lambda_i^{\mathbb{A}} + \lambda_i^{\mathbb{S}}} = \frac{\mu_i^j + \lambda_i^j}{\mu_i^j + \lambda_i} \tag{A.4}$$

The metric $\overline{p^j_{u,i}}$ as well represents the probability of O_i being locally accessed or updated before the object become invalid due to updates from the server or any other clients. From (3.3), (A.3), and (A.4), no matter what replacement policy is used, we can deduce,

$$\begin{aligned}
P_{RAP}(t) &= P_{PAP}(t) \\
&= \sum_{\forall i|O_i \in C'(t)} p^j_{a,i}\overline{p^j_{u,i}} \\
&= \sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}\setminus\{O_r(t)\}} p^j_{a,i}\overline{p^j_{u,i}} \\
&= \sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}\setminus\{O_r(t)\}} \frac{\mu^j_i}{\mu^j}\frac{\mu^j_i+\lambda^j_i}{\mu^j_i+\lambda_i} \\
&= \sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}\setminus\{O_r(t)\}} \frac{(\mu^j_i+\lambda^j_i)\mu^j_i}{(\mu^j_i+\lambda_i)\mu^j} \\
&= \sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}} \frac{(\mu^j_i+\lambda^j_i)\mu^j_i}{(\mu^j_i+\lambda_i)\mu^j} - \frac{(\mu^j_r+\lambda^j_r)\mu^j_r}{(\mu^j_r+\lambda_r)\mu^j} \\
&= \frac{1}{\mu^j}\left[\sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}} \frac{(\mu^j_i+\lambda^j_i)\mu^j_i}{\mu^j_i+\lambda_i} - \frac{(\mu^j_r+\lambda^j_r)\mu^j_r}{\mu^j_r+\lambda_r}\right]
\end{aligned} \tag{A.5}$$

We can drive the follow equality when URP is exercised,

$$P_{RAP+URP}(t) = P_{PAP+URP}(t) = \frac{1}{\mu^j}\left[\sum_{\forall i|O_i \in C(t)\cup\{O_a(t)\}} \frac{(\mu^j_i+\lambda^j_i)\mu^j_i}{\mu^j_i+\lambda_i} - \frac{(\mu^j_{URP}+\lambda^j_{URP})\mu^j_{URP}}{\mu^j_{URP}+\lambda_{URP}}\right] \tag{A.6}$$

By definition, O_{URP} has the lowest gain factor among all the cached and newly fetched objects.

$$\begin{aligned}
GF^j_{URP} &\equiv \min_{\forall i|O_i \in C(t)\cup\{O_a(t)\}} \left(GF^j_i\right) \\
\Rightarrow \frac{(\mu^j_{URP}+\lambda^j_{URP})\mu^j_{URP}}{\mu^j_{URP}+\lambda_{URP}} &\equiv \min_{\forall i|O_i \in C(t)\cup\{O_a(t)\}} \left(\frac{(\mu^j_i+\lambda^j_i)\mu^j_i}{\mu^j_i+\lambda_i}\right)
\end{aligned} \tag{A.7}$$

From (A.5), (A.6) and (A.7), we can draw the following equality,

$$\begin{aligned} P_{RAP+URP}(t) &= P_{PAP+URP}(t) \\ &= \frac{1}{\mu^j}\left[\sum_{\forall i|O_i\in C(t)\cup\{O_a(t)\}} GF_i^j - \min_{\forall i|O_i\in C(t)\cup\{O_a(t)\}}\left(GF_i^j\right)\right] \\ &\geqslant \frac{1}{\mu^j}\left[\sum_{\forall i|O_i\in C(t)\cup\{O_a(t)\}} GF_i^j - GF_r^j\right] \\ &= P_{RAP}(t) = P_{PAP}(t) \end{aligned} \tag{A.8}$$

From (A.8), it is clear that when URP is employed as the replacement policy and RAP/PAP for access policy, at each replacement, the probability of effective hits is higher than or equal to any other policy. So, we assert that URP maximizes the probability of guaranteed effective hits at each replacement for both RAP and PAP access mechanism when updates are injected from both the clients and the server.■

From Theorem 5, we can postulate Corollary 5 and Theorem 6.

Corollary 5 *When the data updates are injected by the clients as well as the server, URP minimizes the expected cost of data access at each replacement for both RAP and PAP access mechanism.*

Theorem 6 *In a cache system where all updates are injected from the clients as well as the server, in the long run, URP gives optimal guaranteed effective cache hits for both RAP and PAP.*

The corollary and the theorem are proved in Appendix B and Appendix C, respectively.

Corollary 6 *When the data updates are injected by the clients as well as the server, URP minimizes the expected cost of data access in the long run for both RAP and PAP access mechanism.*

Proof of the corollary is presented in Appendix D.

Appendix B

Cost Optimality of the Proposed Policies

Here, we prove Corollary 1, 3 and 5.

Proof: With the RAP cache access scheme, when an access results in an effective cache hit, the client and the server exchange a request and acknowledgement message only (Fig. 3.3(a)). Otherwise, i.e., if the access causes a cache miss or an invalid cache hit, the client sends the server a request message, and the server responds to the client by fetching the requested data object (Fig. 3.3(a)). If the cache is full, the server also forwards a replacement decision (Fig. 3.3(b)). Therefore, for RAP,

$$\begin{aligned} C_{RAP}(t) &= P_{RAP}(t)(C_{req} + C_{ack}) + (1 - P_{RAP}(t))(C_{req} + C_{obj}) \\ &= C_{req} + C_{obj} + P_{RAP}(t)(C_{ack} - C_{obj}) \end{aligned} \tag{B.1}$$

In all practical applications, the cost of transmitting a simple message, such as an acknowledgment, is much smaller than the cost of transmitting an entire object, i.e., $C_{ack} \ll C_{obj}$. If simple messages are more or as expensive as fetching data objects, deployment of caches for strongly consistent applications would be more expensive in terms of both computing and communication resources. Thus, $C_{ack} - C_{obj} < 0$, but $P_{RAP}(t) \geqslant 0$ and $P_{RAP+URP}(t) \geqslant 0$. We know that $P_{RAP}(t)$ is maximized when URP is used, i.e., $P_{RAP+URP}(t) \geqslant P_{RAP}(t)$ (from Theorem 1 when updates are injected from the server only, from Theorem 3 when updates are injected from the clients only, or from Theorem 5 when updated are injected from both the server and the clients). Therefore, when URP is

applied, following properties hold:

$$\begin{aligned} C_{RAP+URP}(t) &= C_{req} + C_{obj} + P_{RAP+URP}(t)(C_{ack} - C_{obj}) \\ &\leqslant C_{req} + C_{obj} + P_{RAP}(t)(C_{ack} - C_{obj}) = C_{RAP}(t) \end{aligned} \tag{B.2}$$

In PAP, in case of an effective cache hit there is no need for communication in between the client and the server. Therefore, the cost of access resulting from the replacement at the t-th access is:

$$C_{PAP}(t) = (1 - P_{PAP}(t))(C_{req} + C_{obj}) \tag{B.3}$$

Again, (from Theorem 1, 3 and 5), $C_{PAP}(t)$ is minimized when URP is employed to make replacement decisions, i.e., $P_{PAP+URP}(t) \geqslant P_{PAP}(t)$. Thus,

$$\begin{aligned} &C_{PAP+URP}(t) \\ &\quad = (1 - P_{PAP+URP}(t))(C_{req} + C_{obj}) \\ &\quad \leqslant (1 - P_{PAP}(t))(C_{req} + C_{obj}) = C_{PAP}(t) \end{aligned} \tag{B.4}$$

■

Appendix C

Long Term Optimality of the Proposed Policy

Here, we prove Theorem 2, 4 and 6.

Proof: Let the content of a cache be C up to time $t1$. Let object O_{a1} be accessed at $t1$, and there is another replacement policy called OPT making a different decision than URP. The objects replaced by URP and OPT are O_{URP1} and O_{OPT1}, respectively. By definition, $GF_{URP1} \leqslant GF_{OPT1}$. It has been proven (in Theorem 1 for system where updates are injected from the server only, in Theorem 3 for systems where updates are injected from the clients only, and in Theorem 5 for systems where updated are injected from both the server and the clients) that the guaranteed hits for URP are higher than or equal to any other replacement policy, including OPT. Assume that $P_{URP}(t1) > P_{OPT}(t1)$. To makeup the loss and outperform URP, let at a later time $t2 > t1$, when O_{a2} is accessed, OPT makes another replacement decision, where URP continues with the existing cache content. Thus, $P_{URP}(t1) = P_{URP}(t2) < P_{OPT}(t2)$. At time $t2$, object O_{OPT2} is decided to be replaced. The events are shown in Fig. C.1. Next, we prove the theorem by contradiction and show

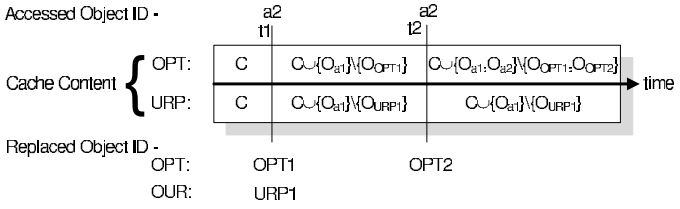


Figure C.1: Replacement in OUR and other policies

that OPT replacement policy cannot exist.

$$\begin{aligned} & P_{OPT}(t2) > P_{URP}(t1) \\ \Rightarrow \quad & \sum_{\forall i|O_i \in C} GF_i + GF_{a1} + GF_{a2} - GF_{OPT1} - GF_{OPT2} \\ & > \sum_{\forall i|O_i \in C} GF_i + GF_{a1} - GF_{URP1} \\ \Rightarrow \quad & GF_{a2} > GF_{OPT2} + GF_{OPT1} - GF_{URP1} \end{aligned} \tag{C.1}$$

Here, $GF_{OPT1} > GF_{URP1}$ and thus, $a2 \neq OPT2$ and the replacement at $t2$ involves in an eviction. It is clear that the following equation must hold:

$$O_{OPT2} \in C \cup \{O_{a1}\} \backslash \{O_{OPT1}\} \tag{C.2}$$

We consider three possible cases for choosing O_{OPT2}.

Case 1, where $O_{OPT2} \in C \backslash \{O_{OPT1}, O_{URP1}\}$: URP would make an eviction and accommodate O_{OPT2}, as $C \backslash \{O_{OPT1}, O_{URP1}\} \subseteq C \cup \{O_{a1}\} \backslash \{O_{URP1}\}$.

Case 2, where $O_{OPT2} \equiv O_{URP1}$: From (C.1), we can derive $GF_{a2} > GF_{OPT1}$. As $O_{OPT1} \in C \cup \{O_{a1}\} \backslash \{O_{URP1}\}$, URP would evict $O_k \equiv \min_{\forall i|O_i \in C \cup \{O_{a1}\} \backslash \{O_{URP1}\}} O_i$, where either $k \equiv OPT1$ or $(GF_{OPT1} \geqslant GF_k \Rightarrow GF_{a2} > GF_k) \wedge (k \neq OPT1)$.

Case 3, where $O_{OPT2} \equiv O_{a1}$: With similar argument of case 2, it can be shown that URP would make a replacement decision to evict some $O_k \in C \cup \{O_{a1}\} \backslash \{O_{URP1}\}$ to accommodate O_{a2}.

Therefore, it is not be possible that OPT makes an eviction decision to accommodate O_{a2} at $t2$ while satisfying (C.1), and at the same time, URP does not also accommodate O_{a2} by evicting one of the cached objects. Hence, a policy like OPT does not exist. Using the same argument, it can be shown that there exists no sequence of replacements (by another replacement policy) which results in higher guaranteed effective hits than URP.■

Appendix D

Long Term Cost Optimality of the Proposed Policies

Here, we prove Corollary 2, 4 and 6. The proof is similar to the short term cost optimality proof presented in Appendix B.

Proof: From (B.1), we can drive that

$$\begin{aligned}\widetilde{C_{RAP}} &= \widetilde{P_{RAP}}(C_{req} + C_{ack}) + (1 - \widetilde{P_{RAP}})(C_{req} + C_{obj}) \\ &= C_{req} + C_{obj} + \widetilde{P_{RAP}}(C_{ack} - C_{obj})\end{aligned} \tag{D.1}$$

Here, $\widetilde{C_{RAP}}$ and $\widetilde{P_{RAP}}$ are used to denote long term cost and probability of effective hits, respectively, when RAP is exercised. From the discussion of Appendix B, we know that $C_{ack} - C_{obj} < 0$, $\widetilde{P_{RAP}} \geqslant 0$ and $\widetilde{P_{RAP+URP}} \geqslant 0$. From Theory 4, 2 and 6, we further know that $\widetilde{P_{RAP+URP}} \geqslant \widetilde{P_{RAP}}$. Therefore, when URP is applied, following properties hold:

$$\begin{aligned}\widetilde{C_{RAP+URP}} &= C_{req} + C_{obj} + \widetilde{P_{RAP+URP}}(C_{ack} - C_{obj}) \\ &\leqslant C_{req} + C_{obj} + \widetilde{P_{RAP}}(C_{ack} - C_{obj}) = \widetilde{C_{RAP}}\end{aligned} \tag{D.2}$$

For PAP, we can drive the following from (B.3):

$$\widetilde{C_{PAP}} = (1 - \widetilde{P_{PAP}})(C_{req} + C_{obj}) \tag{D.3}$$

Here, $\widetilde{C_{PAP}}$ and $\widetilde{P_{PAP}}$ are the notations for long term cost and probability of effective hits, respectively, when PAP is exercised. Again, (from Theorem 2, 4 and 6), $\widetilde{C_{PAP}}$ is minimized when URP is employed to make replacement decisions, i.e., $\widetilde{P_{PAP+URP}} \geqslant \widetilde{P_{PAP}}$. Thus,

$$\begin{aligned} \widetilde{C_{PAP+URP}} &= (1 - \widetilde{P_{PAP+URP}})(C_{req} + C_{obj}) \\ &\leqslant (1 - \widetilde{P_{PAP}})(C_{req} + C_{obj}) = \widetilde{C_{PAP}} \end{aligned} \tag{D.4}$$

■

Appendix E

Probability of No Updates

Let there be two independent Poison Processes with rate μ_1 and μ_2. In this appendix, we will find the probability to happening an event of the first process before an event of the second process.

Let $N_1(t)$ and $N_2(t)$ be the number of events from the first and second process, respectively, in between time t_0 and $t_0 + t$.

We have,

$$P[N_1(t) = n] = \exp^{-\mu_1 t} \frac{(\mu_1 t)^n}{n}$$

and,

$$P[N_2(t) = n] = \exp^{-\mu_2 t} \frac{(\mu_2 t)^n}{n}$$

The inter-arrival time of two consecutive events from the first process, X, follows exponential distribution with parameter μ_1, i.e.,

$$f_X(x) = \mu_1 \exp^{-\mu_1 x}$$

Therefore, probability of no event from the second process, in between two events from the first process

$$
\begin{aligned}
&= \int_{-\infty}^{\infty} P[N_1(x) = 0 | X = x] f_X(x) dx \\
&= \int_{-\infty}^{\infty} \exp^{\mu_2} \mu_1 \exp^{-\mu_1 x} \\
&= \frac{\mu_1}{\mu_1 + \mu_2}
\end{aligned}
$$

To use the above equation to drive the probability of an object not being updated, we may consider the first process being the combination of access processes and the second process being the combinations update processes to the concerned object.