Mansoura University
Faculty of Engineering
Mansoura Engineering Journal

Cache Policies for Smartphone in Flexible Learning: A Comparative Study

سياسات التخزين المؤقت للهواتف الذكية في التعليم المرن: دراسة مقارنة

Ibrahim Y. Abdel-Baset, *Mohamed A. Mohamed, Ahmed Sh. Samra* and Ahmed Abou-Taleb

KEYWORDS:
Mobile learning, Cache, Cache techniques, Cache policies, Cache replacement policy, Smartphone, Performance metrics.

***الملخص العربي*:-إن التطور السريع للتكنولوجيا غيرت وجه التعليم وخاصة استخدام الأجهزة النقالة التى لا تخلو منها يد أى طالب. مصطلح "الأجهزة النقالة" يشمل أنواع مختلفة من الأجهزة (مثل الهواتف الذكية، الهواتف المحمولة، المساعد الرقمي الشخصي (PDA)، وما إلى ذلك). أمل استفادة من هذه الأجهزة في التعلم المتنقل لإتاحة التعلم في أي مكان وزمان. قد يكون هناك بعض العقبات التي تحول دون استخدام هذه الأجهزة، مثل الإتصال المستمر بشبكات الإنترنت, عرض النطاق الترددي اللاسلكية, وطاقة البطارية للجهاز النقال. كفاءة تخزين البيانات من أهم الطرق للحد من نقل البيانات لاسلكيا في أنظمة المعلومات المتنقلة. في هذه الورقة نقدم نظرة عامة على معظم تقنيات التخزين لاختيار الأفضل واستخدامة لتطبيقات التعليم النقال على الهواتف الذكية, عن طريق استخدام معايير الأداء الخاصة بالتخزين المؤقت والتي يمكن اعتمادها لتقييم نوعية سياسة التخزين**

***Abstract*—The rapid evolution of technology has changed the face of education especially usage of mobile device that aren't devoid of any student hand. The term "mobile devices" covers many different kinds of devices (e.g. smart phones, cell phones, personal digital assistant (PDA), tablets, netbooks, etc.). Hoping take advantages of these devices in mobile learning which allow learning anywhere and anytime. There may be some obstacles to use these devices, such as wireless bandwidth and client's battery power**

Data caching is an appropriate technique for reducing wireless data transmissions in mobile information systems. In this paper we present overview of most popular cache techniques which use performance metrics that might be adopted to assess the quality of caching policy.

Received: (16 June, 2016) - revised: (10 July, 2016) - accepted: (25 August, 2016)

Ibrahim Y. Abdel-Baset,Communications Engineering Dept., Faculty of Engineering, Mansoura University, Egypt
Mohamed A. Mohamed,Communications Engineering Dept., Faculty of Engineering, Mansoura University, Egypt
Ahmed Sh. Samra,Communications Engineering Dept., Faculty of Engineering, Mansoura University, Egypt
Ahmed Abou-Taleb,Communications Engineering Dept., Faculty of Engineering, Mansoura University, Egypt

I. INTRODUCTION

LEARNING via mobile devices is widely accepted by the learner community. Learners are interested in using all available mobile learning resources through mobile phones and personal digital assistants (PDAs) to access information anytime and anywhere. According to Molenet, mobile learning can be broadly defined as the exploitation of ubiquitous handheld technologies, together with wireless and mobile phone networks, to facilitate, support, enhance and extend the reach of teaching and learning. Mobile learning provides an educational environment in which learners can learn without any limitation of time, place, or device; there by realizing a more creative and learner-centered educational process can take place in any location [1].

Mobile devices and mobile databases have become common these days. Caching frequently accessed data on the client side is an effective technique for improving performance in a mobile environment. But the caching schemes in mobile environment need to be different from those in wired networks, due to many reasons: (i) wireless networks have a limited bandwidth; (ii) downstream bandwidth can be much higher than the upstream one and the clients usually can't directly talk to each other, and (iii) mobile clients also limited power and could be disconnected for long periods of time [2].

Therefore, any of the cache invalidation schemes have to be energy efficient and support long and frequent disconnections. Wireless bandwidth and client's battery power are the two scarcest resources, which can be measured by packet efficiency and power efficiency. Packet efficiency means the ability of the algorithm to minimize the total number of packets sent on wireless link. Power efficiency refers to the ability to minimize the energy spent by the client that is running the algorithm [3]. The system performance is measured in terms of access efficiency, which is minimizing the period of time from when a mobile computer issuing a data request until the time the data item is received. These are the criteria that must be considered in cache management for mobile environment [2]. Mobile devices have limited amount of internal storage (usually around 128GB at most). And after the operating system and applications, the remaining space for digital content is much less than this nominated value [4]. Some of the devices can extend their capacity by using a memory card, but the capacity of these cards is also limited. The speed of a wireless connection is low in comparison to a wired connection. The highest wireless speed is often limited by the use of the Fair User Policy (FUP) by the mobile connection provider. The FUP restricts the quantum of the downloaded data in a period of time. In addition, the speed of a wireless connection can vary. The newest connection technologies are not available everywhere, but mobile users wish to access their data as fast as possible. So far, users download the same data repeatedly; we can use a cache to increase system performance. A cache is an intermediate component which stores data that can be potentially used in the future. While using a cache; the overall system performance is improved. The cache is commonly used in database servers, web servers, file servers, storage servers, etc. [5].

A number of caching policies have been discussed and their performances have been tested in an attempt to minimize several cost metrics such as hit ratio, byte hit ratio, average latency and total power. A good a comparative study that describes several cache replacements polices can be found in. This study classifies cache policies into categories. The focus of our work is on evaluating cache replacement policies that aims at improving cache hit ratio, which is the most general metric to evaluate the performance of a caching system.

In the literature, there are some studies regarding early towards finding a collection of algorithms that have a profound impact on the performance of the network, many caching and replacement algorithms have been proposed. Zeitunlian et al. proposed a cache replacement strategy, the Least Unified Value strategy (LUV) to replace the Least Recently Used (LRU) that Scalable Asynchronous Cache Consistency Scheme (SACCS) [6]. Wong et al. claimed that there are a sufficient numbers of good policies, and further proposals would only produce minute improvements so that the focus should be fitness for purpose rather than proposing any new policies, and identifies the appropriate policies for proxies with different characteristics such as proxies with a small cache, limited bandwidth, and limited processing power [7]. Waleed Ali et al. presented a survey and discussed some studies that take into consideration impact of integrating both web caching and web prefetching together [8]. Bžoch et al. presented shortcoming of caching algorithms, proposes LFU-SS and LRFU-SS as new caching policies testing them with commonly used caching policies like LRU and LFU [5].

The remaining of this paper are organized as follows: Section-2 describes the caching techniques where classified into; Partitioned storage, distributed cache architecture and cache replacement policy were divided into three categories: simple, sophisticated and hybrid algorithms, Section-3 shows most common performance metrics to evaluate the performance of represented cache policies, Section-4 details how our experimental setup referring to software simulator, generated tested database and methodology, Section-5 shows in details evaluation results according to experimental setup section, and finally, concluding recommendations from the obtained results.

II. CACHING TECHNIQUES

Web caching is one of the most successful solutions for improving the performance of Web-based system. In Web caching, the popular web objects that likely to be visited in the near future are stored in positions closer to the user like client machine or cache server. Thus, the web caching helps in reducing Web service bottleneck, alleviating of traffic over the Internet and improving scalability of the Web system. Fig. 1 shows a simple local caching architecture. When a user requests a content which is already cached in the local cache server, the local cache server sends the cached content to the user without requesting the content from the original remote server. Consequently, local caching can both increase the users' quality of experience and can decrease the network traffic [7].

Caching technique has attractive advantages to Web participants, including end users, network managers, and content creators: (i) decreases user perceived latency, (ii) reduces network bandwidth usage, (iii) reduces loads on the origin servers, and (iv)saving clients' battery power [9]. Caching techniques can be divided into three categories: partitioned storage, distributed caching, and cache replacement policy.

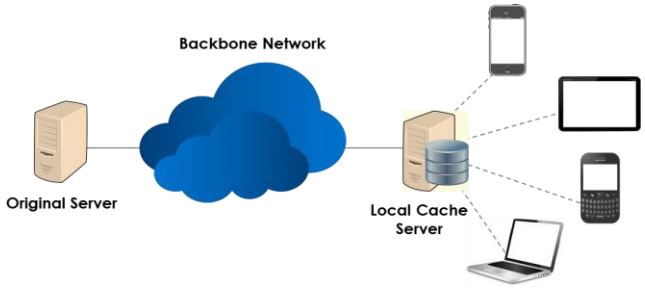

Fig. 1. Simple local caching architecture

A. *Partitioned storage*

Mobile clients do not have enough capacity for storing a huge amount of the data in general. Partitioning provides

scalability and reliability for applications. When the total size of the data is greater than the heap in any single member, partitioned regions can provide the needed data management [2]. An effective solution is to partition the large file size as streaming media into small chunks and only cache parts of file in local cache. Partial caching techniques can be classified into two types: (i) Time-based partial caching such as prefix caching and arbitrary segments and (ii) Bandwidth-based partial caching as Video Staging [10]. These caching techniques include segmentation of streaming objects, dynamic caching, and self-organizing cooperative caching [11]

B. Distributed cache server architecture

Another approach to implement large-scale cache is distributed caching. In distributed caching, no intermediate caches are set up and only caches at the edge of the network cooperate to serve each other's misses. In the year of 2001, it was realized that distributed caching were becoming popular with the emerging of new applications that allow distributions of web pages, images, and music since distributed caching has lower transmission times than distributed caching due to the fact of most traffic flows through less congested lower network levels [12]. The benefits of distributed caching are twofold: they allow balancing the server load during busy period and they allow scaling the caching system's capacity. Most of distributed file systems (DFS) are developed for wired clients and do not support mobile devices. Accessing files from mobile devices requires algorithms which take into account changing communication channels caused by user's movement. DFS that are widely used were made before mobile clients have been spread, and it is difficult to develop mobile client applications now. None of current DFS e.g. Andrew File System (AFS), Network File System (NFS), Coda, InterMezzo, BlueFS, CloudStore, GlusterFS, XtreemFS, dCache, MooseFS, Ceph and Google File System does not have suitable clients for mobile devices [13].

.

C. Cache replacement policy

Due to the limited cache space, suitable replacement policy is required to decide which content should be replaced for a newly arrived content when cache is full. The decision which objects to remove is made by a caching policy algorithm (also called replacement policy or removal policy). The problem is to find the value of an object, and whether it should be cached, without dramatically increasing computation and communication overheads. There are many factors of caching that influence the replacement process include: recency, frequency, size, cost of fetching the object, modification time (time of last modification), and expiration time (time when an object gets stale and can be replaced immediately) [14]; as shown in Fig. 2

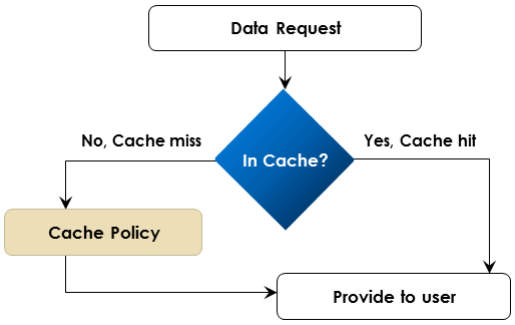

Fig. 2. Cache Policy

A page replacement policy looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself. This was the basis why we require a page replacement algorithm. Cache replacement policy can be divided into three categories: simple, sophisticated and hybrid algorithms. Table1. Presents a simple comparison between them, and they can be defined in details as follows:

1- Simple Caching Algorithms

Simple caching algorithms do not use any statistics or additional information for data replacement. For replacement decision, they usually employ other mechanisms. Examples of simple caching algorithms are Random (RAND) Algorithm, First-In First-Out (FIFO) Algorithm, FIFO with 2nd chance (FIFO2) Algorithm and CLOCK Algorithm.

RAND is a simple replacement policy which chooses data to be replaced based on random selection [15]. While this strategy provides a very easy implementation as it only requires a random or pseudo-random generator and only O (1) additional amount of work per page replacement, it has many drawbacks such as it doesn't take user's behavior into account or even take any advantage of any temporal or spatial localities.

FIFO is another low-overhead paging algorithm which chooses data to be replaced based on the oldest in the cache. Where ordering cache data reference in a queue, oldest data at front and newest at end. When new data are come to full cache, the data from queue header are evicted, and the new data are inserted to queue tail [16]. While this strategy provides simple implementation as it has a low overhead, it has many drawbacks same as of RAND policy – FIFO policy does not take user's behavior into account, might throw out useful data may be used soon, requires a queue Q to store data references in the cache, Data are enquired a de-queue operation on Q to determine which one to evict.

FIFO2 is a modification of the FIFO caching policy.

FIFO2 stores the data units in a queue. In contrast to FIFO, FIFO2 stores a reference bit for each data unit in the queue. If a cache hit occurs, the reference bit is set to 1. When a replacement is needed, the oldest unit in the cache with a reference bit set to 0 is replaced and the reference bit of the older units is set to 0 at the same time [17].

CLOCK is another simple replacement policy in which all page frames are visualized to be arranged in form of a circular buffer that resembles a clock. The hand of the clock is used to point to the oldest page in the buffer. Each page has an associated reference bit that is set whenever the particular page is referenced. The page replacement policy is invoked in case of a page miss, in which case the page pointed to by the hand, i.e. the oldest page is inspected. If the reference bit of the page is set, then the bit is reset to zero and the hand is advanced to point to the next oldest page. This process continues till a page with reference bit zero is found. The page thus found is removed from the buffer and a new page is brought in its place with reference bit set to zero [18]

2. Sophisticated Caching Algorithms

The drawbacks of RAND and FIFO taken into account where sophisticated algorithms employ some statistical information about data in the cache: frequency of the accesses which used by least frequently used (LFU) algorithm, and recency of last use of data which used by least recently used (LRU) algorithm.

LFU is a sophisticated replacement policy that chooses data to be replaced based on data frequently used. Where using a counter for each data block exists that increases every time the data block is accessed. When new data are come to full cache, the data which is less frequent to use is replaced by the new incoming one [19]. While this strategy has long sighted, it has many drawbacks such as the data blocks in the cache that have been accessed for many times in a short period of time remain in the cache, and cannot be replaced [20].

LRU is a sophisticated replacement policy which uses temporal locality of the data [13]. This algorithm works on the time-stamp. When new data are come to full cache, the data that have not been accessed for the longest time will not be used in the near future, can be replaced by the new incoming one [19]. While this strategy provides high adaptability, it has many drawbacks such as the data block can be replaced even if the block was accessed for many times. It does not consider about the frequency of the objects, i.e., how many times that particular object was referenced. In the recently used one, according to the Belady's anomaly the more objects the memory has in the recent time, the fewer object faults/ hit miss a program will get [21].

LRU-MIN is an enhanced of LRU to minimize the replacement. This algorithm keeps a smaller size of object in the cache. If there is any object with Size S in the cache, then follows the LRU algorithm to evict least recent used object. If there is no such an object which is having size S in the cache, then this algorithm evicts the object of size S/2 in the least recently used order [22].

LRU-Threshold is similar to LRU, but with a subtle difference is that, an object which is largest than a threshold size is not inert into the cache [23].

LRU-K is an algorithm keeps the timestamps of the last K accesses to the data block. When new data are come to full cache, LRU-K counts so-called Backward K-Distance which leads to mark data block to replace. LRU-2 is an example of LRU-K which remembers last two access timestamps for each data block. It replaces then the data block with the least recent penultimate reference [24].

MRU Most Recently Used algorithm evicts the most recently used document from the cache. This algorithm is used where we have to access the historical information [25].

SLRU Segmented LRU algorithm is similar to LRU-K but seems easier to implement. Where the cache is divided into two segments: the protected segment and the probationary segment. On a miss, data is then pended on the MRU part of the probationary segment. Hits are added to the MRU part of the protected segment. As the protected segment has a definite size, adding a line into the protected segment pushes the LRU line of the protected segment to the MRU part of the probationary segment. This method avoids flooding the cache with data that will not be reused, because the protected segment contains lines which have been accessed at least twice. The best results were obtained when the size of the protected segment is around 80% of the cache. It performs around 3-4% better than LRU for a cache size of 0.5 Mb [26].

LFU-SS Least Frequently Used-Server Statistics algorithm works similarly as regular LFU, but with a use server and local statistics for replacement decision. Where the database module of the server maintains metadata for the files stored in the DFS. The metadata records contain items for storing statistics. These statistics are read and write hits per file, and global read hits for all files in the DFS. When a user reads a file from the DFS, the READ_HITS counter is increased, and sent to the user. When a user wants to write the file content, the WRITE_HITS counter is increased. Both of these counters are provided for each requested file. The GLOBAL_HITS counter is provided on demand. When new data are come to full cache, the read hits counter for a new file is initialized to one (the file has been read once). The idea of LFU-SS is that firstly calculate the read hits counter from the statistics from the server. If the new file in the cache is frequently downloaded from the server, the file is then prioritized in comparison to a file which is not frequently read form the server. For cached files in LFU-SS, use three operations: inserting new file into cache, removing file from the cache, and updating file read hits. LFU-SS prevent ageing files in the cache by division the READ_HITS client by two. Calculation of priorities for replacement is not computationally demanding because of relatively low number of units in the cache. LFU-SS achieves up to 11% of improvement over LRU in smaller cache capacities [13].

3. Hybrid Caching Algorithms

The drawbacks of LRU and LFU replacement policies result in hybrid algorithms. These algorithms use a

combination two or more algorithms to get better results in cache hit ratio.

2Q Two Queues algorithm is an improving the detection of real hot data and remove cold data faster from the main memory. This algorithm works by two separate queues. One is maintained as LRU queue serves for so-called hot data blocks that have been accessed more than once, and the other as FIFO queue serves for so-called cold data blocks that have been referenced only once. When new data comes to full cache, it stores in FIFO-queue. If the same data block is accessed for the second time, it moves to the LRU-queue. 2Q algorithm gives approximately 5% improvement in hit ratio over LRU [27].

MQ Multiple Queues algorithm is a replacement policy which chooses data to be replaced based on hit's count priority. Every queue has its own priority. The data blocks with lower hit's count are stored in lower priority queue. If the number of hit's count reaches the threshold value, the data block is moved to the tail of queue with higher priority. When the replacement is needed, the data blocks from the queue with the lowest priority are replaced [28].

ARC Adaptive Replacement Cache algorithm is similar to 2Q replacement policy but it uses two LRU-queues. These queues maintain the entries of recently evicted data blocks. The two queues remember twice the number of pages that would fit in the cache. The ARC algorithm dynamically balances recency and frequency. It is simple to implement and has low computational overhead while performing well across varied workloads [29]. This algorithm performs as well as the fixed replacement policy with the optimal p but it is dynamic and no parameter needs to be tuned before and hence should perform the same way through all workloads and cache parameters, contrary to the other policies presented above [30].

LRFU Least Recently/Frequently Used algorithm is spectrum of policies that subsumes LRU and LFU Policies. Unlike the LRU and LFU policies that consider frequency or recency only, the LRFU policy takes into account both the frequency and recency of references in its replacement decision. Furthermore, unlike LRU_K policy that considers only the last K references to a block. The LRFU policy associates a value with each block. This value is called Combined Recency and Frequency (CRF) and considers all the past references to a block to appraise the likelihood that the block may be used in the near future. Each reference to a block in the past contributes to this value and a reference's contribution is determined by a weighing function F(x) is calculated which considers the data objects reference time span from in the past to the current time [31].

LRFU-SS Least Recently Frequently Used-Server Statistics algorithm is other hybrid caching replacement policies; it is a combination of LFU-SS and standard LRU. Which chooses data to be replaced based on final priority selection. Where computes the priority of LRU and LFU-SS for each file in the cache. The final priority of LRU and LFU-SS interval 0 to 65535, where higher number means that the file is more suitable for storing in the cache. The priority value for the LFU-SS algorithm is calculated by using linear interpolation between the greatest and the lowest read hits values. The priority value for the LRU algorithm is calculated from the timestamp for last access to the file. When new data are come to full cache, the file with the lowest final priority is evicted, and the new data are inserted. LRFU-SS achieves up to 10% of improvement over LRU and LFU in larger cache capacities. While this strategy provides, it has many drawbacks such as it needs to recalculate priorities for all cached units every time one cached unit is requested, and needs to reorder the heap of the cached files because of changes of these priorities [5].

LRU/LFU Least Recently Used/Least Frequently Used algorithm is hybrid caching used both recency and frequency. Where this policy included a threshold value i.e. TSD to evict the historical object which are not been used by the long time. Frequency division (FD) is used to calculate the average frequency. When a new object is entering, it checks the least time and check the priority with the frequency division. If the priority is greater than frequency division, then look for the second smallest timestamp. If the difference between both the time-stamp is greater than TSD, then removes the first least document else if not, then calculate the priority with the frequency division. If the priority is greater than the frequency division than removes the first document else second [19].

III. PERFORMANCE METRICS

The appropriate caching policy will be selected based on comparing the performance of different caching policy algorithms to be adopted in mobile learning devices. This selection is based on two goals: to minimize the costs of counting the priority of data block in the cache; it should be also taken into account that the mobile devices are limited capacity. The speed of connection from the mobile device to the remote server can vary and to increase cache hit ratio, by decreasing the network traffic.

TABLE 1.
COMPARISON OF CACHE REPLACEMENT TECHNIQUES

Caching Technique	Brief Description	Available replacement Policy	Performance to LRU
Simple	This technique does not use any statistics or additional information to evict web object.	RAND, FIFO, FIFO2, CLOCK.	RAND 22% worse [32] FIFO 12-20% worse [32]
Sophisticated	This technique employs some statistical information about data in the cache to evict web object.	LFU, LRU, LRU-MIN, LRU-Threshold, LRU-K, MRU, SLRU, LFU-SS.	LRU-k Around 50% better for very large database buffers [24] SLRU Around 3-4% better [26]
Hybrid	This technique based on a combination of two or more algorithms to evict web object.	2Q, MQ, ARC, LRFU, LRFU-SS, LRU/LFU.	2Q5-10% better [27] LRFU-SS achieves up to 10% of improvement over LRU [13]

To evaluate the performance of the cache, performance metrics are being used. Some common metrics that are used to quantify the performance such as are Hit rate, byte hit rate, bandwidth saved, delay saving ratio are most commonly used which can be defined as follows

3.1- Hit Rate (HR)

The Hit Rate (HR) is the percentage of all requests object which are found in the cache instead of transferred from the requested server [33], this can be expressed as:

$$\mathrm{HR} = \sum_{i\in \mathrm{R}} h_i \Big/ \sum_{i\in R} f_i \qquad \textbf{(1)}$$

Where h_i is the number of object hit for an object i, f_i is the total number of request for the object i, and R is the set of objects which accessed.

3.2- Byte Hit Ratio (B_{HR})

This metric is similar to hit rate, except it emphasizes the total saved bytes by caching certain objects. It is the percentage of all data that is transfer straight from the cache rather than from requested server [34], described as:

$$B_{\mathrm{HR}} = \sum_{i\in \mathrm{R}} S_i h_i \Big/ \sum_{i\in R} S_i f_i \qquad \textbf{(2)}$$

where S_i is the size of the object *i*.

3.3- Bandwidth Saved (BS)

The bandwidth saved tries to quantify the decrease in the number of bytes retrieved from the original servers reducing the amount of the bandwidth consumed. This is directly related to byte hit ratio [35].

3.4- Delay Saving Ratio ($\mathrm{T_{DS}}$)

The measure the latency (the interval between the time the user requests for a certain content and the time at which it appears in the user browser) of fetching an object [35]. It has been defined as the sum of penalty times of the hits over the sum of the penalty times from all the requests [36], can be expressed as:

$$T_{\mathrm{DS}} = \sum_{i\in \mathrm{R}} d_i h_i \Big/ \sum_{i\in R} d_i f_i \qquad \textbf{(3)}$$

Where d_i is the delay which occurs to retrieve the object from server

3.5- Average Downloads Time (T_{AD})

Due to the inconsistency in download time in account of traffic variations, performance results based on this metric may vary [33].This metric tries to find average download time by:

$$T_{\mathrm{Ad}} = \sum_{i\in \mathrm{R}} d_i \left(1 - h_i / f_i\right) \Big/ \|R\| \qquad \textbf{(4)}$$

Where $\|R\|$ is the size of R.

These performance metrics are the heart of web caching algorithm where are used to evaluate the performance of the replacement algorithms with respect to object which are present and requested in memory of cache, the saved bytes due to no retransmission, and the decrease in latency to retrieve an object which is requested.

IV. Experimental Results

The performance of all presented cashing replacement policies will be assessed and compared using Cache Simulation tool which have three parts with Server, Client and Request generator. Eleven cashing replacement algorithms have been tested and evaluated: RND, FIFO, Standard LFU, LRU, LRU-K, MRU, 2Q, MQ, LFU-SS, LRFU, and LRFU-SS. These cashing algorithms provided the best performance metrics among all existing web cashing algorithms. Every caching policy has its own coefficients. We have made 10,000 random requests on files for 500 files with random size between 5KB and 5GB. We have used cache hit ratio, cache read hit counts, saved byte, and data transfer decrease needed to transfer the files as performance indicators. Following subsections, 5.1, 5.2 and 5.3 discuss more simulation setup setting.

4.1. Software Simulator

Cache simulator (CS) is a tool which serves for evaluation of caching policies and consistency control algorithms. It develops to prevent the main disadvantage of testing caching policies in a real environment, where it used to evaluate cache polices based on simulation setting as request input method, file sizes, cache capacities, and average network speed and consistency control [5]. CS consists of three parts: Server, Client and Request generator described as follows: (i) Server represents storage of files collection. Each file is represented by a unique ID and size in bytes. Additionally, the server stores a number of read and writes requests for each file. When a client demands a file, all the metadata are provided; (ii) Client is an entity which requests files from the server and uses the evaluated caching algorithm. During the simulation, the client receives requests for file access from the Requests generator. The client increases the counter of requested bytes by the size of the file and looks into its cache for a possible cache hit. If the file is found in the cache, the number of cache read hits is increased. If the file is not in the cache, the file is downloaded from the server and stored in the cache. At the same time, the counter maintaining the number of transferred bytes is increased by the size of the requested file, and (iii) Requests generator is an entity which knows the files' ID from a server, and generates requests for these files.

4.2. Generated Test Database

Before the simulation can start, several input parameters

have to be set. Firstly, we set request input method. The requests can be generated randomly by using one of the random generators–uniformly random, uniformly random with preference, Zipf random or Gaussian random. We can set varied parameters for each generator. Each of these generators may require additional parameters which can be also set. We used a Gaussian random generator for a simulation with parameters. Secondly, we have created 500 files with random size between 500KB and 5GB on the server side. The size of files respects the fact that mobile clients usually accesses smaller files from the remote storage. We have made 10,000 random requests on files where some of the files are prioritized and some other files are accessed less often. Reflecting the limited capacity of mobile devices; we used cache sizes ranging from 8MB to 1024MB. For each cache size, the cache polices will be evaluated separately. The simulation setting lists and parameters

TABLE 2.
PARAMETERS SETTING FOR SIMULATION

Description	Setting
Number of files	500
File popularity	Gaussian Random
Number of requests	10,000
File size	500KB ~ 5GB
File bit rate	80 Mbps
Cache sizes	8~1024MB
LRU-K	K =3, correlated reference period=7
2Q	50% of cache capacity for FIFO
MQ	Life time=100, Out queue capacity=10, Queue count=5
LRFU	P =2, λ=0.045
LRFU-SS	K1=0.35, K2=1.1

4.3 Methodology

The performance of any cache policy depends on one or more parameters; some of them are related to the training process of their caching policy such as RND, FIFO, Standard LFU, LRU, MRU, and LFU-SS. And other parameters related to the testing process such as LRU-K (with k=3, correlated reference period=7), 2Q (with 50% of cache capacity for FIFO), MQ (with life time=100, out queue capacity=10, queue count=5), LRFU (with P=2, λ=0.045) and LRFU-SS (with K1=0.35, K2=1.1). Parameters of caching policies were obtained while changing their correlated periods and the best results were selected for each policy [13].

Performance indicators are used to evaluate the performance of the replacement algorithms. Four metrics will be used: (i) cache hit ratio (HR); (ii) Cache Read Hit Counts (represents the number of requests which have been served by the cache), we have observed the read hits count, and then we have computed read hit ratio. On the other side, the cache read hits count deals only with the count of the files in the cache that were found in the cache. (iii) Saved Byte, we use whole file as a basic caching unit. Hence, the policy with the best read hits count does not have to be the best caching policy in saving data traffic because of variable file size, and (iv) data transfer decrease needed to transfer the files (which is also the number of bytes transferred without usage of a cache) as performance indicators.

V. EVALUATION RESULTS

In this subsection, we give the results of the simulations. We have observed performance metrics which applies for caching algorithms according to experimental setup section. We show the results as summaries tables for Read Hit Ratio, Saved Byte and Data Transfer Decrease Ratio. On the other hand, we framed result as figures for Read Hit Counts, Saved Byte Ratio and Data Transfer Decrease.

Overall, results in this experiment show that LFU-SS achieves up to 2% improvement in saving network traffic in smaller cache sizes over other caching policies. LRU-K achieves up to 1% improvement in higher cache sizes. In the experiment, we have observed the read hits count, and then we have computed read hit ratio. Read hits count represents the number of requests which have been served by the cache. The experiment used cache sizes from 8MB to 1024MB. Table 3 summarizes Read Hit Ratio, and Fig. 3 depicts read hit count for each of the implemented algorithms. For each simulated caching policy, we have had the same scenario of accessed files

TABLE 3.
READ HIT RATIO VS. CACHE SIZE

Caching Policy	Cache Size [MB]							
	8	16	32	64	128	256	512	1024
RND	3.08	6.31	10.98	18.43	28.97	43.49	63.71	90.36
FIFO	3.12	6.32	10.97	18.19	29.58	43.92	62.8	89.49
LFU	10.58	21.05	24.98	33.34	42.69	54.88	71.67	91.25
LRU	3.19	6.59	12.22	20.88	33.13	48.25	67.76	91.42
LRU-K	3.19	7.66	21.57	33.41	43.73	55.53	72.05	91.34
MRU	2.42	3.09	4.94	7.17	12.6	24.27	45.36	84.94
2Q	7.43	17.45	25.2	34.17	43.67	54.57	72	91.25
MQ	9.45	16.97	21.3	27.59	36.9	50.09	68.27	91.42
LFU-SS	10.62	19.89	26.06	33.98	41.54	52.04	68.71	91.49
LRFU	8.55	16.23	25.1	34.14	44.3	55.69	71.18	91.41
LRFU-SS	8.28	10.87	17.14	25.52	35.98	49.87	68.45	91.48

For different cache sizes we have different better policy. LFU-SS has the best result for cache size 8M, 32M and 1024M. For cache sizes from 128MB to 256M, LRFU is a better choice. Standard LFU is better at 16M, 2Q is better at 64M and LRU-K is better at 512M

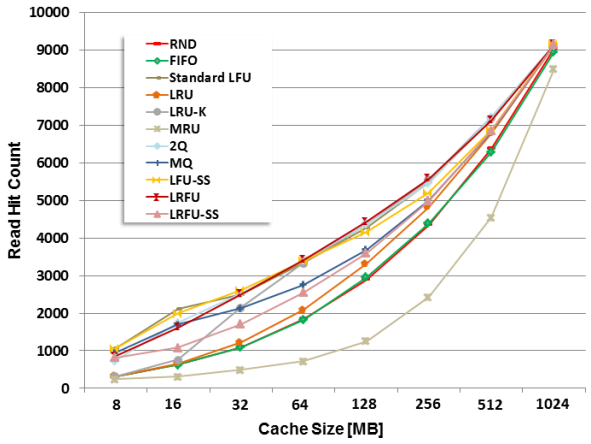

Fig. 3. Read Hit Counts vs. Cache size

Next, we measured the saved bytes. Table 4 summarizes the data saved byte for different caching policies. As shown is Fig. 4, represent saved byte ratio counts with different cache size.

TABLE 4.
SAVED BYTE VS. CACHE SIZE

Caching Policy	Cache Size [MB]							
	8	16	32	64	128	256	512	1024
RND	655	1378	2417	4040	6422	9719	14432	20646
FIFO	675	1395	2434	4010	6577	9870	14177	20426
LFU	2130	3635	4565	6551	9050	12213	16251	20859
LRU	691	1478	2713	4595	7314	10739	15252	20899
LRU-K	689	1670	4442	6660	9426	12543	16320	20878
MRU	490	631	1014	1540	2826	5462	10199	19427
2Q	1569	3398	4969	7204	9354	12164	16281	20859
MQ	2026	3451	4440	5825	8070	11090	15381	20897
LFU-SS	2158	3792	5427	7287	9142	11623	15471	20911
LRFU	1836	3232	4870	7088	9418	12437	16165	20881
LRFU-SS	2033	2583	3883	5321	7811	11120	15404	20905

LFU-SS has the best result in saved bytes for cache sizes from 8MB to 64MB and 1024MB. For cache sizes from 128MB to 512MB, LRU-K is a better choice. On the other side, the cache read hits count deals only with the count of the files in the cache that were found in the cache. We use whole file as a basic caching unit. Hence, the policy with the best read hits count does not necessarily the best caching policy in saving data traffic because of variable file size.

Saved Byte Ratio for each of the implemented algorithms can be depicted in Fig (4). Consecutively, for smaller cache size (8MB to 64MB) LFU-SS can achieve up to 11% improvement over commonly used LRU cache policy. For larger cache sizes (128MB to 512MB), LRU-K achieved up to 10%in saved byte ratio

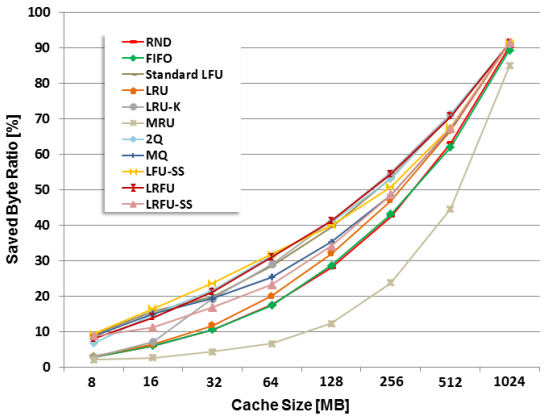

Fig. 4. Saved Byte RatioCounts vs. Cache size

Finally, we measured the data transfer decrease. The total size of transferred files was 22.8GB. Table 5 summarizes different caching policies with its data transfer decrease ratio (DTDR). Fig. 5 shows data transfer decrease for different cache policies.

TABLE 5.
DATA TRANSFER DECREASE RATIO VS. CACHE SIZE

Caching policy	Cache Size [MB]							
	8	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
RND	97.136	93.979	89.439	82.35	71.94	57.538	36.944	9.796
FIFO	97.048	93.905	89.365	82.478	71.264	56.875	38.06	10.756
LFU	90.692	84.115	80.056	71.378	60.46	46.641	28.996	8.865
LRU	96.979	93.542	88.144	79.923	68.043	53.078	33.363	8.69
LRU-K	96.989	92.704	80.593	70.899	58.815	45.199	28.696	8.781
MRU	97.857	97.243	95.566	93.271	87.652	76.137	55.44	15.124
2Q	93.143	85.154	78.289	68.524	59.132	46.855	28.867	8.865
MQ	91.148	84.922	80.6	74.55	64.741	51.544	32.798	8.699
LFU-SS	90.569	83.432	76.287	68.163	60.059	49.216	32.407	8.64
LRFU	91.976	85.879	78.719	69.032	58.849	45.663	29.373	8.77
LRFU-SS	91.116	88.714	83.033	76.752	65.872	51.416	32.699	8.664

For cache sizes from 8MB to 64MB and 1024MB, LFU-SS has the best result in data transfer decreasing ratio where can achieve up to 10%. For cache sizes from 128MB to 512MB, LRU-K achieves up to 10% of improvement over LRU in large cache sizes.

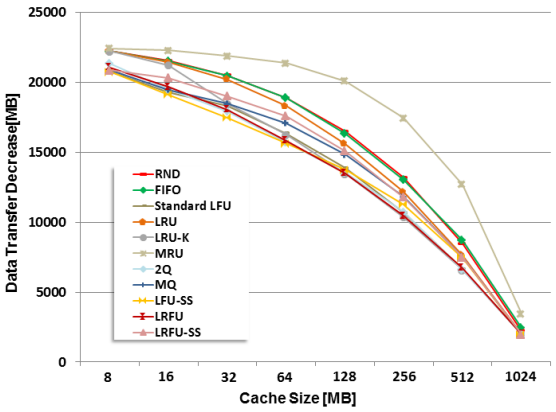

Fig. 5. Data Transfer Decrease vs. Cache size

LFU-SS has the best result in decreasing network traffic for cache sizes from 8MB to 64MB and 1024MB. For cache sizes from 128MB to 512MB, LRU-K is a better choice.

Experimental simulations show the following: (i) Policy with the best read hits ratio is not necessarily the best one in decreasing data traffic; (ii) For larger cache size all policies near to have the same values that clearly at cache size 1024MB.We have the worst values up to 23% - according to commonly used LRU cache policy- belong to MRU policy; (iii) LRU-SS is the best cache algorithm for cache sizes (8MB, 16MB, 32 MB, 64MB and 1024MB) which achieve up to 11% of improvement over LRU cache policy, and (iv) The other best cache policy is LRU-K for cache size (128MB, 256MB and 512MB), that achieve up to 10% of improvement over LRU cache policy. While using LRU-K with a cache size of 512MB, it saved up 71% of the network traffic.

VI. Conclusion

This article presented a comparative study of the performance of different cashing replacement policies for caching files in mobile devices. Our goals in selecting caching algorithms for mobile learning were to decrease network traffic, and minimize the power consumption of mobile battery. These two goals were set because of the varying network connection quality of mobile devices caused by the movement of the user, and because of the limited performance of the mobile devices. The comparison of caching policies made in the section experimental results shows that the introduced algorithms act better in comparison to commonly used caching policies like LRU and LFU. For smaller cache size, LFU-SS is suitable caching policy; for larger cache size, LRU-K is better choice.

In our future work, we will implement cache and caching policy for smartphone client platform applications. Using two algorithms, one for smaller cache size and the other for lager cache size may have complexity. In our future work, we intend to develop an algorithm for all cache sizes. Wireless data transmissions in mobile information systems have bounded bandwidth and client's limited battery power that makes the connection to the cellular networks unstable and expensive. Wherefore, we will implement so-called online/offline operations. In this case, the user can access online for updates and access for cached files even after disconnection as offline

References

[1] Hashemi, Masoud, et al. "What is mobile learning? Challenges and capabilities." Procedia-Social and Behavioral Sciences 30 (2011): 2477-2481.
[2] Rathore, Rooma, and Rohini Prinja. "Caching Schemes in Mobile Databases."
[3] Imielinski, Tomasz, and S. Viswanathan. "Adaptive wireless information systems." Proceedings of the Special Interest Group in Database Systems (SIGDBS) Conference. 1994.
[4] CNET Corporation,"Wireless mobile storage expander roundup: Frequent travelers, you'll want one of these", 2014. [Online]. Available: http://www.cnet.com/
[5] Bžoch, Pavel, et al. "Design and Implementation of a Caching Algorithm Applicable to Mobile Clients." Informatica 36.4 (2012).
[6] Zeitunlian, Aline, and Ramzi A. Haraty. "An Efficient Cache Replacement Strategy for the Hybrid Cache Consistency Approach." World Academy of Science, Engineering and Technology 63 (2010): 268-273.
[7] Wong, Kin-Yeung. "Web cache replacement policies: a pragmatic approach."Network, IEEE 20.1 (2006): 28-34.
[8] Ali, Waleed, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. "A survey of Web caching and prefetching." International journal of advances in soft computing and its application 3.1 (2011): 18-44.
[9] Davison, Brian D. "A web caching primer." Internet Computing, IEEE 5.4 (2001): 38-45.
[10] Tang, Xueyan, Jianliang Xu, and Samuel T. Chanson, eds. Web content delivery. Vol. 2. Springer Science & Business Media, 2006.
[11] Jinlong, Wu. "Analysis of the Performance of CacheReplacement Policies for aVideo-on-Demand System." (2013).
[12] Rodriguez, Pablo, Christian Spanner, and Ernst W. Biersack. "Analysis of web caching architectures: hierarchical and distributed caching." Networking, IEEE/ACM Transactions on 9.4 (2001): 404-418.
[13] Bzoch, Pavel, et al. "Towards caching algorithm applicable to mobile clients."Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on. IEEE, 2012
[14] Rathore, Roma, and Rohini Prinja. "An Overview of Mobile Database Caching."CiteSeerX, doi 10.1.100 (2007): 9481.
[15] Reed, Benjamin, and Darrell DE Long. "Analysis of caching algorithms for distributed file systems." ACM SIGOPS Operating Systems Review 30.3 (1996): 12-21.
[16] Swain, Debabrata, et al. "Analysis and Predictability of Page Replacement Techniques towards Optimized Performance." (2011): 12-16.
[17] Draves, Richard. "Page Replacement and Reference Bit Emulation in Mach."USENIX Mach Symposium. 1991.
[18] Chavan, Amit S., et al. "A Comparison of Page Replacement Algorithms."ACSIT International Journal of Engineering and Technology 3.2 (2011).
[19] Kapil Arora, Dhawaleswar Rao Ch . "Web Cache Page Replacement by Using LRU and LFU Algorithms with Hit Ratio: A Case Unification." International Journal of Computer Science and Information Technologies 5.3 (2014): 3232 – 3235.
[20] Press, Avi, et al. "Caching Algorithms and Rational Models of Memory."
[21] Harish, Polanki, and Wilson Thomas. "A Novel Approach to Enhance the Efficiency of Distributed Cooperative Caching in SWNETs." Journal of Computer Science & Systems Biology 8.1 (2015): 45.
[22] Kuppusamy, P., B. Kalaavathi, and S. Chandra. "Optimal Data Replacement Technique For Cooperative Caching In Manet." ICTACT Journal on Communication Technology 5.3 (2014).
[23] Vakali, A. I. "LRU-based algorithms for Web cache replacement." Electronic Commerce and Web Technologies. Springer Berlin Heidelberg, 2000. 409-418.
[24] O'neil, Elizabeth J., Patrick E. O'neil, and Gerhard Weikum. "The LRU-K page replacement algorithm for database disk buffering." ACM SIGMOD Record 22.2 (1993): 297-306.

[25] Kakde, Vinit A., and Sanjay K. Mishra. "Effective Web Cache Algorithm."International Journal of Electronics, Communication & Soft Computing Science and Engineering (IJECSCSE) Volume 1 (2012).
[26] Karedla, Ramakrishna, J. Spencer Love, and Bradley G. Wherry. "Caching strategies to improve disk system performance." Computer 27.3 (1994): 38-46.
[27] Johnson, Theodore, and Dennis Shasha. "X3: A Low Overhead High Performance Buffer Management Replacement Algorithm." (1994).
[28] Zhou, Yuanyuan, James Philbin, and Kai Li. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches." USENIX Annual Technical Conference, General Track. 2001.
[29] Megiddo, Nimrod, and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." FAST. Vol. 3. 2003.
[30] Damien, Gille. Study of different cache line replacement algorithms in embedded systems. Diss. PhD thesis, KTH, 2007.
[31] Lee, Donghee, et al. "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies." IEEE transactions on Computers 12 (2001): 1352-1361.
[32] Al-Zoubi, Hussein, Aleksandar Milenkovic, and Milena Milenkovic. "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite." Proceedings of the 42nd annual Southeast regional conference. ACM, 2004.
[33] Joy, Preetha Theresa, and K. Poulose Jacob. "Cache replacement policies for cooperative caching in mobile ad hoc networks." arXiv preprint arXiv:1208.3295(2012).
[34] ElAarag, Hala, Sam Romano, and Jake Cobb. Web Proxy Cache Replacement Strategies: Simulation, Implementation, and Performance Evaluation. Springer Science & Business Media, 2012.
[35] Obaidat, Mohammad S., and Georgios I. Papadimitriou, eds. Applied system simulation: methodologies and applications. Springer Science & Business Media, 2012.
[36] Cárdenas, L. G., et al. "Analysis of Web-Proxy Cache Replacement Algorithms under Steady-state Conditions." WEBIST (1). 2007.