HDC – Hot Data Caching in Mobile Database Systems

Vijay Kumar, Nitin Prabhu
SCE, Computer Networking
University of Missouri-Kansas City
Kansas City, MO 64110.
{kumarv, npp21c}@umkc.edu

Panos K. Chrysanthis
Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania.
panos@cs.pitt.edu

Index terms: Mobile computing, caching, wireless channel and Prefetching.

Abstract

Data caching is an effective way of improving system performance. This is especially crucial in mobile databases because of the limited bandwidth and slower communication speed of wireless channels. In this paper we present a data caching algorithm, referred to as HDC (Hot Data Caching), and compare its performance with LRU (Least Recently Used) cache replacement scheme through simulation.

1 Introduction

In any data processing system [1] it is crucial to efficiently utilize available resources for achieving high performance and scalability. This is especially true for mobile systems where critical resources (wireless bandwidth, RAM, etc.) are inherently limited and continuous connectivity cannot be taken for granted. In this paper we describe our investigation on data caching in Mobile Database Systems (MDS). Caching stores desired data in the local storage of data processing node to improves data availability and data access time. There are two important activities of a caching scheme: (a) identification of data set for caching and (b) transferring data from the remote storage to the local storage of the processing node. An efficient caching scheme tries to optimize these activities within the constraints of system resources. On wired system there may not be any constraints especially in transferring data from the remote location, however, on MDS the situation is entirely different. In this paper we present a caching mechanism for MDS, called *HDC -Hot Data Caching* and compare its performance with most commonly used LRU (Least Recently Used) cache replacement scheme through simulation. Note that HDC can be used in any distributed system equally efficiently. In this paper we do not discuss the problem of cache consistency and assume that it is maintained by MDS.

2 Mobile Database System (MDS)

We envision an information processing system based on a wireless communication discipline, which we refer to as *Mobile Database System (MDS)*. MDS of the future will perform differently than conventional centralized and distributed database systems and they will impose more constraints and demands on the wireless systems of today for performing transaction management activities efficiently. Transaction processing in MDS will be quite diverse probably with many different transaction models and processing modes.

The architecture of the Mobile Database System (MDS) we are investigating is shown in Figure 1. We have added a number of DBSs (database Servers) to incorporate database processing capability without affecting any aspect of the generic mobile network [5].

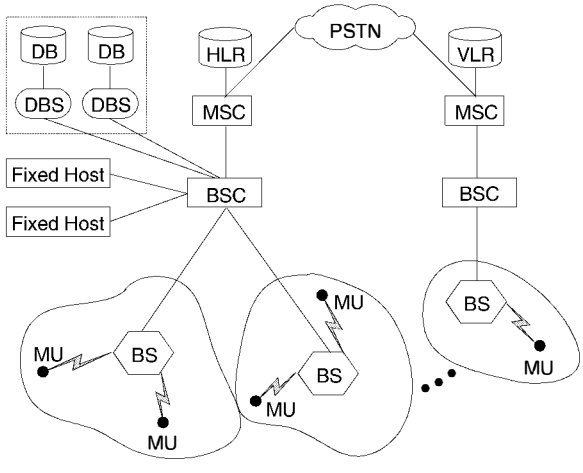

Figure 1. A reference architecture of MDS.

0-7803-8735-X/05/$20.00©2005 IEEE

A set of general purpose computers (PCs, workstations, etc.) are interconnected through a high speed *wired* network. These computers are categorized into *Fixed hosts* (*FH*) and *Base stations* (*BS*) or *mobile support stations* (MSS). A number of mobile computers (laptop, PDAs, etc.), referred to as *Mobile Hosts* (MH) or *Mobile Units* (*MU*) are connected to the wired network components only through *BSs* via wireless channels. A *BS* maintains and communicates with its *MUs* via the air interface [5, 6] and has some processing capability [6].

3 HDC – Hot Data Caching Algorithm

We desire that HDC should be able to satisfy the data needs of maximum number of transactions running at *MUs*. This can be achieved if the desired data sets for the present and also for recent future sets of transactions are identified reasonably accurately and made them available locally.

To achieve very high cache hit rate, we propose to integrate the idea of immediate caching, *prefetching*, and *broadcasting*. With prefetching we cache the future requirements, through broadcasting we create a reservoir of future data in the "channel space" (set of wireless channels available to a cell), and we immediately cache "hot data" through a wireless channel.

3.1 Data Identification for Caching and Transaction Execution

A user initiates transactions through his *MU*. Every *MU* has some upper limit of the number of transactions it can execute concurrently. A *MU* performs the following steps for precisely identifying the data requirements of a transaction.

- Preprocesses the transaction and identifies its data requirement.
- Builds a data structure, refer to as "cache matrix", to store these data.
- Combines the data requirement set just created with the data in the cache using EXCLUSIVE OR ($\bigoplus$) to generate the final data requirement set.

We illustrate the working of the data identification phase of HDC with a simple example. We assume that the size of MUs workload (number of transactions that can be scheduled concurrently) is 10 transactions and the average data item requirement of transactions is 8. D1, D2, ..., represent data items and T1, T2, ..., represent transactions. With the above data cache matrix 1 (Table 1) is created. A "1" in the matrix indicates that the corresponding transaction needs that data item. All 1's in each column is added. A lowest value column sum identifies the data item, which is accessed by least number of transactions. The lowest value column data and the corresponding transactions are removed from the matrix. Thus, in the above matrix D11 and D20 and T1 and T4 are removed, which gives Cache matrix 2 (Table 2).

Table 1. Cache matrix 1

	D1	D8	D10	D11	D15	D20	D22	D30
T1	1	1		1				1
T2			1		1			
T3	1						1	1
T4			1			1		
T5	1	1						
T6	1				1			1
T7		1						
T8			1		1			
T9	1		1					
T10							1	1
Sum	5	3	4	1	3	1	2	4

Table 2. Cache matrix 2

	D1	D8	D10	D15	D22	D30
T2			1	1		
T3	1				1	1
T5	1	1				
T6	1			1		1
T7		1				
T8			1	1		
T9	1		1			
T10					1	1
Sum	4	2	3	3	2	3

The same reduction process is repeated until we get the set, which can fit in the cache. So if we assume a cache size of 4 data items, then the final data set is given in cache matrix 3.

Table 3. Cache matrix 3

	D1	D10	D15	D30
T2		1	1	
T6	1		1	1
T8		1	1	
T9	1	1		
Sum	2	3	3	1

If we assume that initially the cache is empty then the data set for caching can be identified by the following formula.

D1, D10, D15, D30 $\bigoplus$ *current cache data* = *D1, D10, D15, D30*

We refer to the data set obtained by matrix 3 as "hot data" and the rest of the data set (data set required by transactions T1, T3, T4, T5, T7, and T10) is referred to as "warm data". Hot data set is transferred from the DBS and loaded into the cache of the MU and T2, T6, T8, and T9 are scheduled for execution. The other 6 transactions (T1, T3, T4, T5, T7, and T10) are put into the wait queue and a record of warm data (cache matrix 1) is kept in a safe place. Note that the data set of cache matrix 3 contains a subset of data required by the set of waiting transactions, which constitutes prefetching of data.

Cache matrix 3 identifies four transaction, which can be scheduled immediately for execution with a cache size of 4. We refer to the transactions identified by cache matrix (i.e., T2, T6, T8, and T9) as *Active Transactions (AT).* In reality the cache size would quite large and there would be many more transactions that can be scheduled for execution. In this paper we do not elaborate on cache size identification issue. We define AT to be the set of transactions whose data requirements are satisfied by the current cache contents and the MU can execute them concurrently. Note that the waiting transactions whose at least one data item is present in the current cache are not counted in AT but the transaction blocked due to a conflict with AT is counted. We define an upper limit of transactions that MU can execute concurrently and its value depends on MU's processing capabilities and resource availability. Our definition of the upper limit of transactions is similar to the multiprogramming level (MPL) as defined in operating systems. We will, therefore, use MPL to indicate the upper limit of transactions in this paper. Thus, $MPL = AT + Waiting\ transactions$.

New transactions continue to arrive and one of them enters AT set as soon as an AT transaction terminates and exists the system. The other aspect of HDC is to manage transaction execution for maximum throughput, that is how new transactions are scheduled for execution. We incorporate three scheduling schemes (a) immediately schedule the transaction for execution, (b) add it to the waiting queue to be scheduled later, and (c) send the new transaction to the server for execution. These schemes can either be used individually or in an integrated way.

Immediately schedule the transaction for execution: In this scheme a new transaction is scheduled for execution if its desired data set is available in the cache. Although this approaches minimizes cache refreshes, it may severely starve other transactions. Our effort is to eliminate the starvation without affecting the low cache refresh. We explain below how starvation can happen.

At any time there exists a set of waiting transactions, refer to as "WT". In the present scenario AT = {T2, T6, T8, T9} and WT = {T1, T3, T4, T5, T7, T10}. Under this scheme the system continues accepting new transactions and executing them along with AT until a predefined number of transactions have been executed. At this point a new cache matrix is created using the WT set where some new transactions may end up in the waiting queue or a transaction from WT may again be deleted from the next cache matrix and this trend may persist for some time. We argue that in reality such starvation is highly unlikely and present a solution to handle this scenario.

A close observation indicates that a transaction, which accesses rarely accessed data is likely to starve. In a real data processing environment such situation is highly unlikely. It is well known that transactions data items are highly clustered [4, 2] and we believe that the starvation situation is likely to be transient. We have verified this observation in our simulation experiment presented later.

Our aim in MDS is to process majority of transactions with minimum cache refreshes. If a transaction starves within the minimum cache refreshes, then we assume that it cannot be processed by the MU where it originated within acceptable delay, therefore, it should be sent to the server. To implement this scheme we proceed as follows: *If a transaction continues to starve after a predefined number of cache refreshes, then it is sent to the server for processing and the final result is sent back to the MU for dispatching it to the end user.* We want to identify what is the appropriate number of cache refreshes after which starving transactions at MU should be sent to the server for execution.

Add new transaction to the waiting queue to be scheduled later: If the new transaction has at least one data item in the current cache, then it is added to WT list. As soon as AT transactions are terminated a new matrix is created only with a *right* number of new waiting transactions (not for WT transactions because their data set is already known), which never participated in the cache matrix. This cache matrix data and the WT data are combined together again using $\bigoplus$ to identify the hot data to be cached. Starvation is possible in this scheme too and we apply the same solution as described earlier.

Rarely accessed data items may increase the number of waiting transactions. We illustrate this situation with cache matrix Table 4.

Table 4. Cache matrix 4

	D1	D5	D6	D15	D2	D3	D8	D9	D16
T1	1	1	1	1					1
T3	1	1	1	1				1	
T5	1	1	1	1			1		
T6	1	1	1	1		1			
T7	1	1	1	1	1				
Sum	5	5	5	5	1	1	1	1	1

All these transactions need one rarely data items D2, D3, D8, D9, and D16. With cache size of five data items only one of the transactions can be executed. Here again the cache manager has to decide which accessed one of these five transactions' data items will be cached. All transaction will eventually be completed, however, on the cost of long wait time.

We argue that very few transactions show such a diverse data access pattern and with a suitable size cache this situation can be efficiently handled. In our simulation experiment we show the effect of rarely accessed data items by transactions.

Send the new transaction to the server: In this scheme if the new transaction has no data item in the current cache, then it is sent to the server immediately for processing. This will minimize its waiting time. The server sends the result of this transaction back to MU.

4 Pulling and Caching Hot and Warm Data in HDC

The cache matrix identifies hot and warm data sets. Our objective is to download hot data immediately using available wireless channels. In MDS it is essential to get the data for caching from the server before MU suffers a handoff. Handoff merely introduces delay in getting the data, it does not stop data acquisition. This delay may affect MDS throughput but does not interfere with the logic of HDC. For this reason we did not simulate handoff process.

Broadcasting: Broadcasting is used if the user population shares common data interest. It reduces the load on the server as it has to broadcast only once for the multiple requests for the same data. This makes the server more scalable when there is a large client population. The server broadcasts data on a single shared channel on which the clients listen. The mobile users send their data requests to the server which is identified by the cache matrix. If the mobile user had sent his request for every transaction then the server would have to many requests which may degrade performance. The cache matrix uses the cache size to limit the data set for executing transactions. Here we give two broadcast strategies for data dissemination.

The server receives data item set requirement from every MU. Let D_i be the data set requirement from MU_i.

RxW method: The server forms the set *BDS (Broadcast Data Set)*= $D_i \bigcup BDS$. The structure of the *BDS* is as follows *BDS* = $\{(d_j$, R, W)$\}$ Where d_j is the data item requested, *R* is the number of requests for the data item d_j and *W* is the wait time of the first request for the data item d_j. Broadcasting can be done using *RxW* [8] method. The data item selected for the next broadcast is the one which has maximum value of *R*W*. When data item d_j is broadcasted its corresponding entry from the BS set is removed. The mobile users have to continuously monitor the channel until all its data required is broadcasted.

Simple Broadcast: In this method the server sequentially broadcasts the complete data set D_i requested by MU_i is broadcasted. The selection of the data set D_i to be broadcasted next is done on FCFS (First Come First Serve) basis. In the broadcast if index is used then the client knows when the to expect its data set and can tune in on the channel at that time. This method avoids the necessity for the client being continuously connected and hence saves energy of mobile device. An optimization to this method would be to broadcast hot data which are common in most of the data sets. The remaining data items from the data set are broadcasted in FCFS manner.

5 Performance Analysis

We analyzed the performance in two steps. In the first step, we measured the behavior of HDC alone and in the second step we compared its with *Least Recently Used* (LRU) cache replacement technique through a detailed simulation experiment. We selected LRU because it is one of the most commonly used cache replacement algorithm and it provides a very high hit rate [7]. In LRU for our comparison we maintain the access history of data item accesses. When the number of blocked transactions is equal to MPL cache refresh occurs. The content of cache is determined as K-most frequently used pages where K is the size of the cache. The set of data item from K-most frequently used pages, which are not currently in the cache, are requested from the server. In our method we determine the content of the cache based on the current transaction workload.

HDC tries to minimize the cache refresh rate by identifying and caching the exact data requirement set of transactions. Consequently the effective wireless channel utilization increases. In our simulation experiment we study the cache refresh rate, the percentage of executed transactions in each cache refresh, throughput, waiting time, and rate of transaction dispatch to servers for execution. Table 5 list the simulation parameter values we used to drive the simulator.

The data I/O time (data transfer rate from disk to memory) depends upon a large number of parameters and the rate is continuously increasing. For our simulation we looked at the data transfer rate of a number of disk units and identified a value of 10ms. A simple formula *Data Transfer per second* = (*Spindle speed*$\div 60\times$ *Sector per track* $\times 512)\div 1,000,000$ can also be used.

We consulted a number of sources to identify a suitable value for message transfer time between MU and the server [**?**, 3, **?**, **?**]. For a 1.54Mbps bandwidth time a GSM frame of 156 bytes will take .811ms to reach the destination [3]. The bandwidth of 1.54Mb is usually quite high and may not always be available. On the basis of these values and considering the size of update dispatched by MU we computed 10ms $\times$ 2 as an approximate time to send a message to the server and to compose and send the final results to MU.

Table 5. Simulation parameters and their values

Simulation parameter	Value
Database size	5000 entities
Maximum transaction size	10 entities
Minimum transaction size	2 entities
Cache size	100 – 350
MU CPU speed	50 MIPS
DBS CPU speed	100 MIPS
No. of inst. to perform a read from cache	1000 inst.
No. of inst. to perform a write from cache	2000 inst.
Time to deliver a message over wired network	5 ms
Message delivery time over wireless network	10 ms
Multiprogramming level (MPL)	5 – 100
Update probability	0.0 – 1.0

Figure 2 illustrates the flow of transactions through the simulation model. New transactions arrive to the "Arrival queue" from where it is scheduled for execution. Initially when the system is empty as soon the first transaction arrives, we cache all its data and schedule it for execution. This is equivalent to creating cache matrix for one transaction. This is continued until the cache is full. The other way is to wait until system has MPL transactions and then create the cache matrix. This approach will unnecessary increase the waiting time and reduce CPU utilization. Any new transaction is scheduled for execution only if AT list has space and the cache has all its data items. If none of its data item is in the cache, then it is immediately sent to "DBS arrival queue" for execution by the DBS. If it has at least one data item in the cache, then it is pushed to the "Blocked queue" from where it goes to "Arrival queue after the cache matrix is recreated. The DBS sends the result of the transaction to MU.

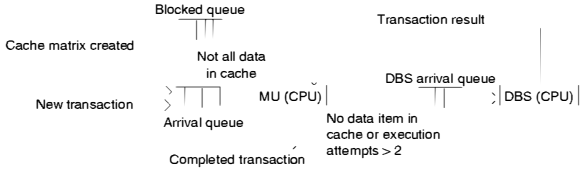

Figure 2. Simulation diagram

5.1 Behavior of HDC

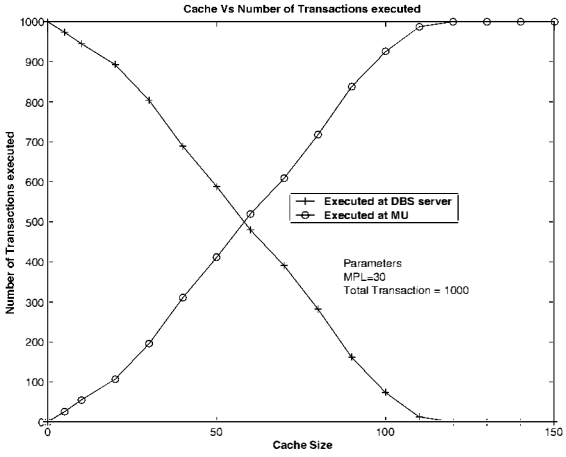

Figure 3. Number of transactions executed at MU and DBS

Number of transactions executed at DBS and MU: Figure 3 shows the number of transactions executed at MU and DBS. We expected this behavior. As cache sizes increases, more and more number of transactions are executed at MU. When the cache size reaches to 200 all transactions find their data items at MU cache and executed there. This observation is also verified by data of Table 6.

Cache refreshes with cache size in HDC: Figure 4 shows how cache refresh rate declines when cache size increases. Note that when MPL=30, number of cache refreshes is higher than when MPL=100. This is because with MPL=100, the next cache matrix is created after 100 transactions have joined the waiting queue and the hot data satisfies more number of transactions. Whereas for MPL=30, cache matrix is created after only 30 transactions are in the waiting queue and to process 1000 transactions the cache is refreshed more number of times. We observe that a cache size of about 350 can handle good size workload with minimum cache refreshes.

Transaction wait time with cache size in HDC: Figure 5 shows the average transaction wait time as cache size changes. The average wait time for MPL=100 is larger than for MPL=30 for all cache sizes. This results complements the results of Figure 4. For MPL=100 cache is not refreshed until 100 transactions joins the waiting queue, where as in MPL=30 each transaction has to wait less. Here again the waiting time reaches to a minimum for a cache size of 350.

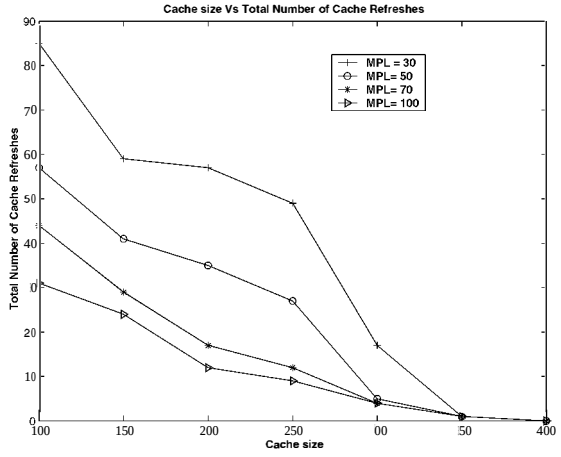

Figure 4. Number of cache refreshes for a cache size

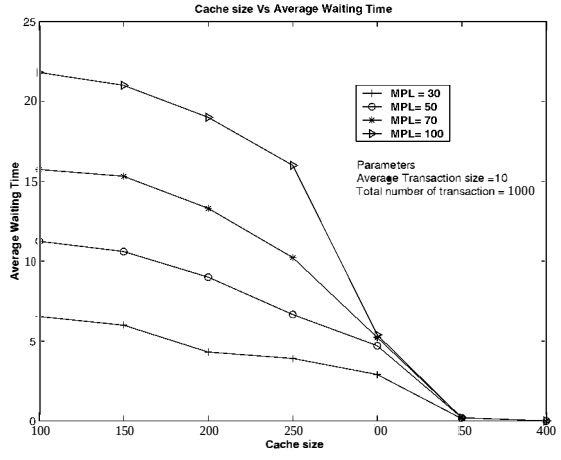

Figure 5. Average transaction wait time with cache size

Transactions executed in a cache refresh in HDC: Table 6 shows what percentage of transactions is completed in each cache refresh for an average transaction size = 10 and MPL = 30. As explained before, in the initial cache, a transaction's required data items are loaded into the cache (cache matrix creation with one transaction). It continues with other incoming transactions until there is no room in the cache. We have fixed an upper limit of cache tries for a transaction. If a transaction is not executed at MU in two tries, then it is sent to the server for execution. Note also that if a new transaction does not have any data item in the cache, then this transaction is also sent to the server for execution. We observe that with cache size of 250 data items about 39% of transaction are executed in the initial attempt (no cache refresh) and the rest are executed in the first cache refresh, thus, no transaction goes to the DBS for lack of cached data items. As the cache size decreases, the percentage of transactions executed in the initial attempt decreases and the percentage of transaction executed in the first cache refresh increases, still most of the transactions are executed in the initial attempt (no cache refresh). We see that even for cache size of 200 data items all transactions are executed with initial cache and one cache refresh and no transaction is sent to the server for execution . The situation begins to change only when the cache size falls to 150 date items but again the percentage in the first cache refresh remains quite low and only 22% of transactions goes to the server. In the worst scenario where the cache size is dropped to 100 data items more and more transactions begins to migrate to first cache refresh attempts and about 60% transactions go to the DBS directly.

Table 6. No. of tries Vs. Percentage of transactions executed

Cache	Initial cache	1 refresh	2 refresh	To server
100	12%	21%	7%	60%
150	23%	48%	7%	22%
200	30%	70%	0%	0%
250	39%	61%	0%	0%

5.2 Performance Comparison

In this section we compare the performance of HDC and LRU algorithms. We maintained identical execution environment for a meaningful comparison. In LRU the least recently used data items were replaced by new data requests. We simulated this algorithm as follows. Initially the cache is empty. As transactions arrive their data items were cached until cache became full. If a new transaction found all its data in the cache, then the transaction is scheduled for execution otherwise the transaction is blocked. A list for frequently accessed data items was maintained and continuously updated as each transaction enters the system. When an active transaction left the system, a new transaction is scheduled for execution and its data items are cached under LRU. The transactions that remain blocked after more than two cache refreshes are sent to the server for execution. Note that in this simulation our goal is to compare the caching scheme with LRU. So we did not simulate broadcasting which depends on the broadcast scheduling and is independent of mechanism used for caching at the mobile client. Since we are focusing only on caching we assumed presence of on demand channel for every mobile client. In

future we plan to study effect of broadcasting strategies discussed in previous section using our caching scheme.

Transaction executed at *MU* and cache size: One of the objectives of HDC is to process maximum number of transactions at *MU*s for improving wireless channel utilization. Figure 6 shows the relation ship between the percentage of transactions executed at *MU* and the cache size. We observe that HDC executes significantly higher percentage of transactions at *MU* compared to LRU scheme. The LRU needs a large size cache (i.e., 350) to execute all transactions at *MU* whereas HDC needs only a cache size of 200 to handle the same workload.

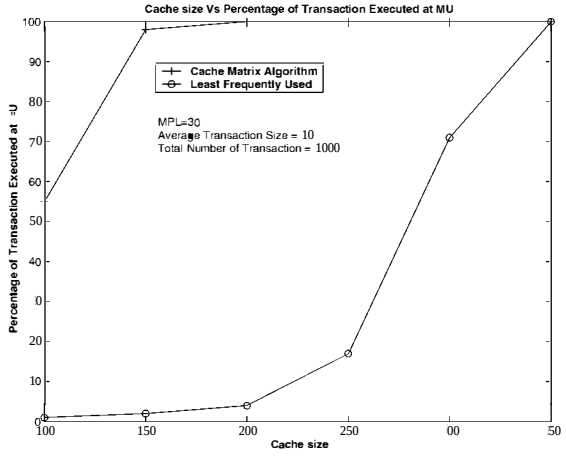

Figure 6. Percentage of transactions executed at MU in HDC and LRU.

Cache refreshes with cache size: Figure 7 shows the relationship between the number of cache refreshes and cache size in these two algorithms. In HDC the number of cache refreshes are significantly less compared to LRU and the decline in cache refreshes is faster in HDC compared to LRU. The difference in cache refreshes between HDC and LRU begins to narrow down only when the cache size = 250. This difference significantly improves channel utilization (uses wireless channels less frequently to satisfy transactions' requirements) in HDC.

Transaction wait time: Transaction wait time also affect system performance. We measured the average wait time of transactions in HDC and LRU. Figure 8 shows the average wait time of transaction in HDC and LRU. In HDC the overall waiting time is significantly less compared to LRU. The average time converges only when the cache size is 350. The wait time begins to narrow down after cache size = 300.

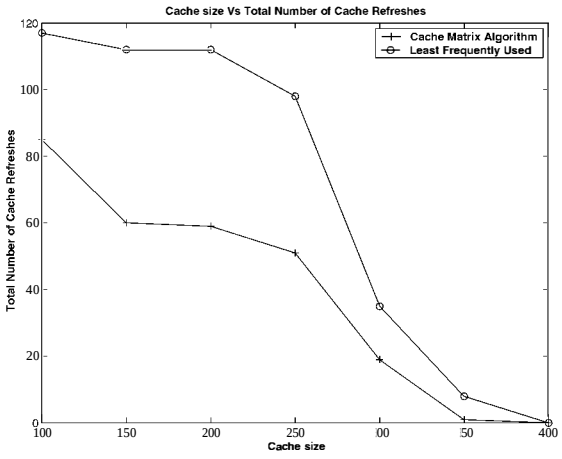

Figure 7. Number of cache refreshes in HDC and LRU

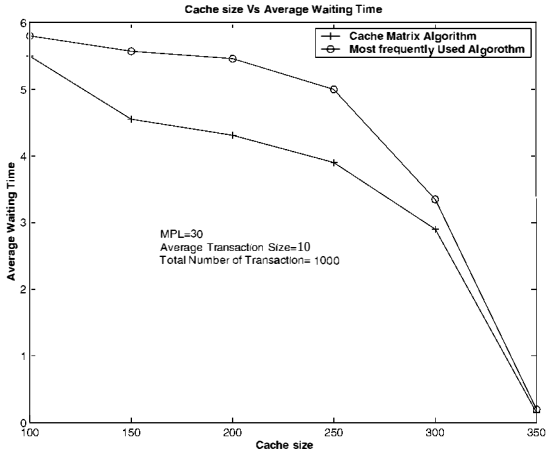

Figure 8. Transaction wait time in HDC and LRU

Throughput: We compared the throughput of HDC and LRU for a number of different cache sizes under different MPLs. All showed similar behavior. Here we have presented results of two cache sizes.

Figure 9 and Figure 10 show how and MPL affects their throughput with cache sizes of 70 and 150 respectively. We see that the throughput of HDC is consistently higher compared to LRU.

At the server also they are subjected to some delay before they are executed and results are sent to the *MU*. We have simulated these delays as follows. We assumed that every cell has 25 wireless channels for communication. If a transaction has to be dispatched to the server for processing,

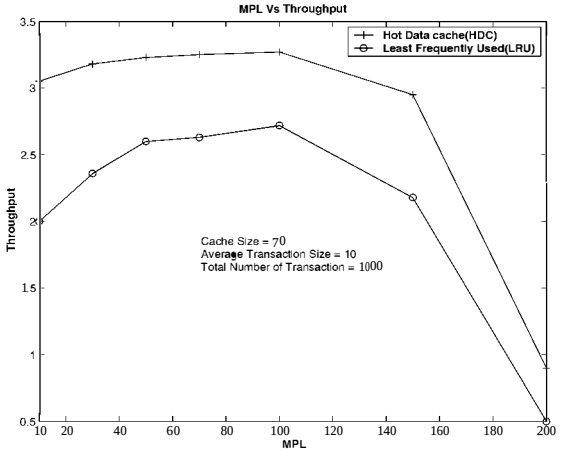

Figure 9. Throughput with different MPLs

then on the average it experiences a delay of 25ms (communication delay 10ms + channel assignment and waiting for the channel 15ms). This value is based on an MPL of 100. Similarly, it experiences a delay of 15ms at the server side (scheduling delay 5ms + communication delay 10ms). The effect of these delays in LRU is only partially compensated by the reduction in execution time at the server because server is much faster than the *MU*.

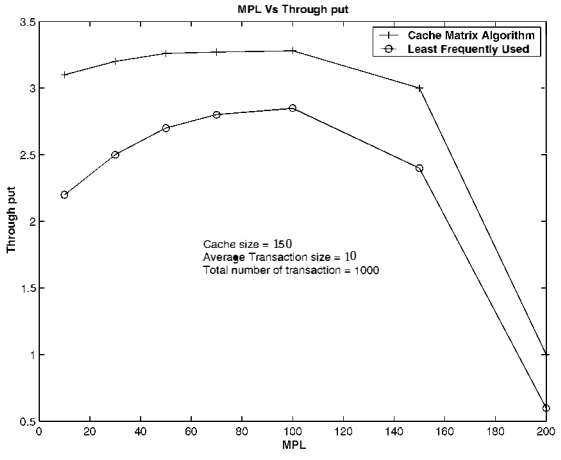

Figure 10. Throughput with different MPLs

The throughput with cache size 70 (Figure 9) is lower than the throughput with cache size of 150 (Figure 10), however, this difference is not significant. This increase is mainly due to the reduction in communication overhead and waiting time because less number of transactions are dispatched when cache size = 150. Note that HDC is not significantly affected mainly because the number of transactions executed at MU with these cache sizes do not differ significantly (Figure 6). This clearly indicates that HDC utilizes the cache far efficiently than LRU does.

6 Conclusions

In this paper we presented a caching algorithm, referred to as HDC (Hot Data Caching), especially suitable for Mobile Database Systems. In addition to HDC, we also presented a number of scheduling schemes for MUs for efficiently utilizing cached data by transactions.

We compared the performance of HDC with Least Recently Used (LRU), which is most commonly used cache replacement algorithm. Our performance comparison through detailed simulation showed that HDC uses fewer wireless channels and cache refreshes compared to LRU for processing the same workload. We also observed that the throughput of HDC is consistently superior to LRU mainly because in LRU majority of transactions are processed at the server. This delays transaction procesing and also increases communication overhead.

References

[1] Chrysanthis, P. K., "Transaction Processing in Mobile Computing Environment". In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–82, October 1993.

[2] Gray, J., and Franco Putzolu, "The Five Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time", Technical report 86.1, February 1986, Tandem Computers.

[3] Vijay K. Garg and Joseph E. Wilkes. "Wireless Personal Communication Systems", PH PTR, 1996.

[4] V. Kumar. "Performance of Concurrency Control Mechanisms in Centralized Database Systems", Prentice Hall, 1996.

[5] R. Kuruppillai, M. Dontamsetti, and F. J. Cosentino. "Wireless PCS", McGraw-Hill, 1997.

[6] M. Mouley and M-B. Pautet. "The GSM System for Mobile Communications", Cell and Sys., 1992.

[7] A. Silberschatz, P. Galvin, and G. Gagne. "Applied Operating System Concepts", John Wiley, 2000.

[8] Demet Aksoy, Michael Franklin, R*x*W: A Scheduling Approach for Large Scale on-demand Data Broadcast, IEEE/ACM Transactions On Networking, Volume 7, Number 6, December 1999.