

Quasi-consistency and Caching with Broadcast Disks

Rashmi Srinivasa and Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, VA
{rashmi, son}@cs.virginia.edu

Abstract. The challenges ensuing from the asymmetric communication capabilities of mobile environments have led to an increased interest in broadcast-based data dissemination. Among the concurrency control (CC) techniques for transactional clients in broadcast environments, BCC-TI has been shown to be more efficient than a traditional technique [1]. We propose two ways of improving CC performance in broadcast environments: caching and a weaker consistency criterion. We demonstrate that caching improves query response time in BCC-TI. We propose a new CC technique called Quasi-TI that enforces a correctness criterion called quasi-consistency [2] — useful when serializability is too expensive to enforce. We introduce a new caching scheme (PIT) and study its effects on Quasi-TI’s performance. Through simulation, we demonstrate the benefits of the proposed techniques.

1 Introduction

In asymmetric communication environments, server-to-client capacity is greater than client-to-server capacity, because of physical attributes (e.g. mobile systems with cellular uplinks) or workload characteristics (e.g. a very large number of clients). Transmission by mobile systems over the air is monetarily expensive due to limited bandwidth [3]. In such environments, pull-based techniques are ineffective, and data dissemination techniques like broadcast disks are popular [4, 5, 6, 7, 8, 9, 10, 11]. Many applications in broadcast environments involve a large number of read-only transactions. In information dispersal systems for stock or auction data, numerous speculators read prices frequently, making efficient query processing essential. Many systems have real-time constraints too (e.g. deadlines to avoid financial loss).

CC techniques have been proposed for transactional clients in broadcast environments. Broadcast CC using Timestamp Interval (BCC-TI) achieves serializability more efficiently than a traditional technique [1]. We explore two ways of improving CC performance in broadcast environments: a weaker consistency criterion, and caching. We propose a new CC technique called Quasi-TI that uses timestamp intervals. Quasi-TI guarantees a weaker consistency criterion (quasi-consistency [2]), which allows a controlled amount of imprecision in the data read by a query. Quasi-TI results in lower query response times than BCC-TI. We propose a new caching scheme (PIT) and study the effects of caching on Quasi-TI and BCC-TI.

2 Related Work

Broadcast-based data dissemination has been studied extensively over a few years [3, 12, 14]. In a broadcast disk model, the server broadcasts all objects in its database in a *broadcast cycle* and the cycle is executed repeatedly. Clients can read the values of the objects as they are broadcast. The model can be that of a flat disk or of multiple disks. The multiple disk model has been extended to allow updates at the server [12].

Typical applications in broadcast environments have clients that execute read-only transactions (queries), and a server that executes update transactions. Databcycle [13] supports transactions, guaranteeing serializability. CC techniques that exploit the semantics of read-only transactions are proposed in [1, 15]. In [15], different queries can observe different orders of update transactions. Quasi-consistency differs from this correctness criterion in that it allows a query to specify the amount and type of imprecision it can tolerate. The BCC-TI [1] scheme guarantees serializability using dynamic adjustment of timestamp intervals [16], and works as follows. During each broadcast cycle, the server stores information on update transactions that committed in that cycle, building a control information table (CIT). The server broadcasts the CIT in the next cycle. At clients, every query has a timestamp interval that is used to record the temporary serialization order induced during execution. When a query reads an object or a CIT, its timestamp interval is adjusted to reflect dependencies between the query and committed update transactions. If the interval becomes invalid (lower bound \geq upper bound), a non-serializable execution is detected and the query is aborted. BCC-TI has been compared to optimistic CC with forward validation adapted to broadcast environments, and performance gains have been shown [1]. Broadcast methods for queries in [3, 17] do not consider real-time constraints.

Semantic-based consistency criteria that are weaker than serializability are presented in [2], in the context of caching data at clients in information retrieval systems in order to improve query response time. In order to reduce the overhead of keeping cache copies consistent, applications allow cache copies to diverge from the central server copy in a controlled manner. Quasi-caching is a technique that allows such controlled divergence. Examples of quasi-caching constraint types include:

1. *arithmetic*: based on difference in values of cache copy and central copy.
2. *version*: based on number of changes between cached and central copies.
3. *delay*: based on time by which cache copy lags behind central copy.

Quasi-caching would be appropriate for a stock trading application. A user interested in stock prices of chemical companies may be satisfied if she reads prices that are within 5% of the true prices (*arithmetic* constraint). On an update, the server decides whether to propagate the value, based on the client's tolerable imprecision level. Propagation-overhead is cut and flexibility in scheduling propagation increases.

3 Quasi-consistency in Broadcast Environments

We propose a new CC technique for broadcast environments called Quasi-consistent Timestamp Intervals (Quasi-TI) based on quasi-caching constraints. Enforcing

quasi-consistency instead of serializability can reduce aborts. Consider a query Q that has read some objects, and requires some more objects before it can commit. If an update transaction U modifies one of the objects that Q has read and then commits, it may be impossible to guarantee serializability by ordering either Q after U or U after Q ; so Q must be aborted. But if U 's modifications leave objects within an imprecision level that Q can tolerate, Q will not be aborted. Let Q_1 be a query at a mobile client, that reads the stock prices of X and Y . Q_1 can tolerate imprecision of up to \$2.50 in the values it reads. U_2 and U_3 are update transactions at the server, and update X by \$3 and \$1 respectively. U_4 is an update transaction that updates Y by \$5. Consider the following events, where r =read, w =write, and c =commit.

Time	8	11	14	16	20
Operations	$w_2(X) c_2$	$r_1(X)$	$w_3(X) c_3$	$w_4(Y) c_4$	$r_1(Y)$

If we enforce serializability, Q_1 proceeds as follows. Q_1 starts with a timestamp interval of $(0, \infty)$. At time 11 when Q_1 reads X , the lower bound of its timestamp interval becomes 8 (last update time of X), because Q_1 must be ordered after U_2 to achieve a serializable execution. At time 14 when X is updated by U_3 , the upper bound of Q_1 's timestamp interval becomes 14 as Q_1 must be ordered before U_3 . At time 20 when Q_1 reads Y , the lower bound becomes 16 (last update time of Y), as Q_1 must be ordered after U_4 . Since the lower bound $>$ the upper bound, Q_1 is aborted.

If we take advantage of Q_1 's tolerable imprecision and enforce quasi-consistency, the sequence of events changes. At time 11, when Q_1 reads X , the lower bound of its interval becomes 8 (last update time of X), as Q_1 must be ordered after U_2 . At time 14 when X is updated by U_3 , Q_1 can ignore the update and still hold a sufficiently precise value of X . The upper bound remains at ∞ . At time 20, when Q_1 reads Y , the lower bound becomes 16 (last update time of Y), as Q_1 must be ordered after U_4 . As Q_1 's timestamp interval is valid, Q_1 commits. This illustrates that a quasi-consistent system can potentially reduce aborts and perform better than a serializable system.

3.1 Server Algorithm

The server stores a database of objects. Update transactions execute at the server and update the values of the objects. Let U be an update transaction, and $WS(U)$ the *write set* of U (set of objects that U modifies). When U commits, the server assigns $TS(U)$ (the *timestamp* of U) to be the current time. For every object d , the server maintains $WTS(d)$, the largest timestamp of a transaction that has modified d and committed. In a broadcast cycle, the server broadcasts every database object d with $WTS(d)$. Periodically, the server broadcasts a control information table (CIT) containing information about recently-committed update transactions, so that clients can check the validity of outstanding queries. When a transaction U commits, the server:

1. Sets $TS(U)$ to the current time.
2. Copies $TS(U)$ into $WTS(d)$, for all d in $WS(U)$.
3. Records $TS(U)$ and $WS(U)$ in the CIT.

The CIT is broadcast at least once (and possibly multiple times) every broadcast cycle. This allows a client to detect conflicts at an early stage and abort invalidated queries. Every time the server broadcasts the CIT, it resets the CIT to an empty table.

3.2 Client Algorithm

Queries execute at mobile clients. Before a query starts execution, it optionally specifies its *quasi-consistency limits*: how much inconsistency the query can tolerate. This information is specified in three forms: [1] imprecision in value of any object: $ImpLimit(Q)$; [2] number of version lags between read version and latest version of an object: $VersionLagLimit(Q)$; [3] time lag between read version and latest version: $TimeLagLimit(Q)$. If any of these limits is exceeded before Q commits, Q is aborted. If Q does not specify one of these limits, it is assumed that Q does not care about the inconsistency introduced in that form. Consider an application where any update can modify d by at most 1.0. If Q can tolerate an imprecision of 3.0 in d , Q can leave $ImpLimit$ unspecified and specify a $VersionLagLimit$ of 3 to ensure correctness.

When a client starts executing query Q, it assigns a timestamp interval of $(0, \infty)$ to Q. Let $LB(Q)$ and $UB(Q)$ be the lower and upper bounds of this interval. When Q requests a read, the client adds the request to the list of objects to be read off the air. The client reads only those objects that are requested by its queries, and also reads every CIT. For every object d that Q reads, Q stores the value of d , $WTS(d)$, and a running estimate (versionLag[d]) of the number of versions by which Q's copy of d lags. When Q reads an object or a CIT, Q's quasi-consistency limits are checked. Q's timestamp interval is adjusted only if its quasi-consistency limits make it necessary to reflect the serialization order induced between Q and committed update transactions. If the timestamp interval becomes invalid, Q is aborted and restarted.

When a requested object d is broadcast, the client reads d 's value and $WTS(d)$. For every query Q that requested the read, the client checks Q's validity as follows:

1. Store the value of d and $WTS(d)$, and set versionLag[d] to zero.
2. Set $LB(Q)$ to $\max(LB(Q), WTS(d))$.
3. If $LB(Q) \geq UB(Q)$, then abort and restart Q.

In other words, the timestamp interval of Q is set to reflect the fact that Q must be ordered after the last transaction that modified d . When a CIT is broadcast, the client checks the validity of each outstanding query Q as follows:

1. For every d that has been read by Q and has since been written by a transaction U in the CIT, versionLag[d] is incremented by the number of such transactions U. If versionLag[d] > $VersionLagLimit(Q)$, then $UB(Q)$ is set to $\min(UB(Q), TS(U))$. If $LB(Q) \geq UB(Q)$, then Q is restarted.
2. For every d that has been read by Q and has since been written by a transaction U in the CIT, the time difference between $TS(U)$ and the read time of d ($WTS(d)$) is calculated. If this difference exceeds $TimeLagLimit(Q)$, then $UB(Q)$ is set to $\min(UB(Q), WTS(U))$. If $LB(Q) \geq UB(Q)$, then Q is restarted.
3. If $ImpLimit(Q)$ is 0, $UB(Q)$ is set to $\min(UB(Q), TS(U))$ for all U in the CIT that have updated an object d read by Q. If $LB(Q) \geq UB(Q)$, then Q is restarted.
4. If $ImpLimit(Q) > 0$, the actual imprecision must be checked the next time the object is broadcast. Therefore, a reread request is enqueued for every such object.

If U leaves objects read by Q within imprecision limits tolerable by Q, then Q does not adjust its timestamp interval in order to serialize after U. But if Q must be ordered after U, Q's interval is adjusted to reflect this order, and Q is aborted if the resulting order is unserializable.

Requests for a reread of object d are processed when the server broadcasts d . For every query Q that requested a reread of d , the client checks Q 's validity as follows:

1. Calculate difference between previously read value of d and new value of d .
2. If difference exceeds $\text{ImpLimit}(Q)$, then set $\text{UB}(Q)$ to $\min(\text{UB}(Q), \text{WTS}(d))$.
3. If $\text{LB}(Q) \geq \text{UB}(Q)$, then restart Q .

4 Caching

Without caching, a client waits an average of half a broadcast cycle to read an object. Caching can improve performance by reducing this waiting time. First, we describe the P-caching (P=Probability-of-access) scheme, where a client caches objects that it will access with a high probability. If the set of queries at a client is static, access probabilities can be computed in the beginning. If the set of queries is dynamic or unknown, the client can keep a running estimate of the access probability per object.

4.1 P-Caching

In P-caching, the entry for a cached copy c consists of the value of the object ($\text{val}[c]$), its timestamp ($\text{TS}[c]$), the version lag that the copy has accumulated ($\text{versionLag}[c]$) and the time lag accumulated ($\text{timeLag}[c]$). The changes to the client's algorithm are as follows. When a query Q requests a read, the client checks if there is a cached copy of the object. If there is a copy (c), the client does the following:

1. If $\text{versionLag}[c]$ and $\text{timeLag}[c]$ are both zero, then Q uses the cached copy c .
2. If $\text{ImpLimit}(Q)$ is unspecified, and $\text{versionLag}[c] < \text{VersionLagLimit}(Q)$, and $\text{timeLag}[c] < \text{TimeLagLimit}(Q)$, then Q uses the cached copy c . Otherwise, Q waits and reads the object the next time the object is broadcast.

When a requested object is broadcast, the client checks if a copy of the object already exists in its cache. If a copy exists, the value and timestamp of the cache entry are updated, and the versionLag and timeLag of the entry are reset to zero. If a copy does not exist, the object is inserted into the cache if there is an available cache slot. If no slot is available, the cache entry with the lowest P-value (probability of access) is selected as a victim to be replaced by the new object.

Every time a CIT is broadcast, the client checks if it has cached copies of any of the modified objects. For every such copy c , $\text{timeLag}[c]$ is set to $(\text{TS}(U) - \text{TS}(c))$, where U is the last transaction that modified the object. $\text{versionLag}[c]$ is incremented by the number of transactions that have updated the object since the last CIT.

4.2 PIT Caching

P-caching uses the criterion of probability of access to decide which objects to cache. In a quasi-consistent system, another criterion becomes important: the tightness of the consistency constraints of a query. The *constraint tightness* of a query is inversely proportional to the amount of imprecision it allows in ImpLimit , VersionLagLimit and TimeLagLimit . Intuitively, if an object is read by queries with

very tight consistency constraints, then caching the object is not very useful because queries may have to read the object off the air anyway, to obtain sufficient precision.

We introduce a new caching scheme PIT (Probability-of-access Inverse constraint-Tightness). Every cached object has a PIT value stored with it. An object’s PIT value is the access probability of the object divided by the maximum of the constraint tightness values of the queries that have read the object. During cache replacement, the victim cache entry is selected to be the one that has the lowest PIT value.

5 Performance Evaluation

We simulated a flat broadcast disk with one server and one client. The client accesses only a subset (ClientObj) of the objects that the server broadcasts, modelling the fact that the server is serving other clients. HotObjects among a total of ServerObj are very frequently updated (with probability HotProb). The simulation runs in units of *bit times* (1 bit time = time to broadcast one bit). We assume that the network delivers objects in FIFO order. Each object is of size ObjectSize bits. An update transaction commits at the server

every UpdArrive Kbit time units. Update transaction size is uniformly distributed between MinUpdSize and MaxUpdSize. An update can change the value of an object by \leq MaxValChange units. The server broadcasts a CIT after every CITcycle objects. For the client, there are ClientHotObj hot objects. The probability of reading hot objects is HotProb. No attempt was made to match up hot spots at server and client. A query arrives every QryArrive Kbit time units, and submits a read every ThinkTime Kbit time units. Query size is uniformly distributed in [MinQrySize, MaxQrySize]. Queries specify ImpLimit, VersionLagLimit and TimeLagLimit when they arrive. These limits are chosen from uniform distributions centred around AvgImp, AvgVLag and AvgTLag. PercNoImp percent of the queries leave ImpLimit unspecified. A query can read objects from a cache of size CacheSize objects. A query is allowed to use a cached copy only if it is known that the copy is sufficiently accurate, given the query’s quasi-constraints. A query is aborted if objects that it has read become too stale before it commits. An aborted query is restarted and run until it commits, even if it has missed its deadline (soft real-time transaction). We simulate a restart by starting a new query. The deadline is (current time + slack factor * predicted execution time), where slack factor is uniformly distributed in [MinSlack, MaxSlack]. Performance metrics are average query response time, percentage of queries restarted and percentage of queries that missed deadlines.

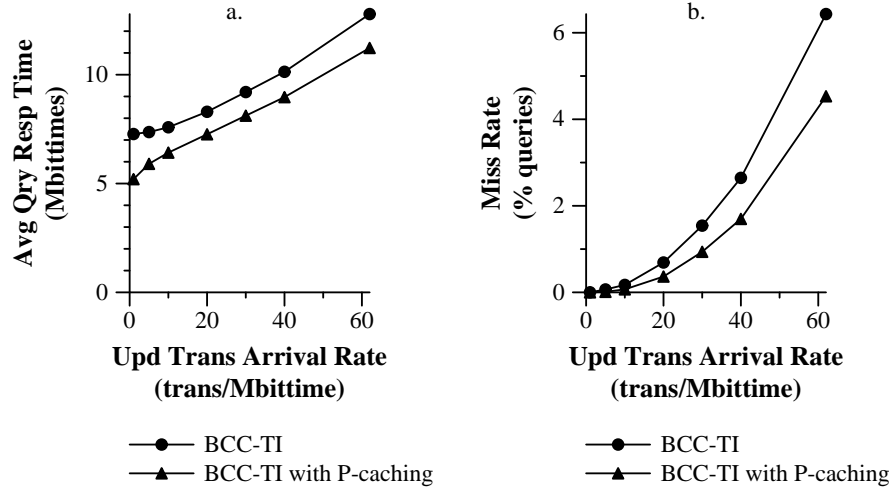
Table 1. Baseline Parameters

Parameter	Value	Parameter	Value
ServerObj	600	HotObjects	200
ClientObj	300	ClientHotObj	100
ObjectSize	8000	UpdArrive	100
QryArrive	132	ThinkTime	64
MinUpdSize	3	MaxUpdSize	4
MinQrySize	3	MaxQrySize	4
MinSlack	2.0	MaxSlack	8.0
CITcycle	100	CacheSize	100
AvgImp	30.0	AvgVLag	6
AvgTLag	3000	HotProb	70
PercNoImp	0	MaxValChange	5.0

5.1 Experiment 1: P-Caching in BCC-TI

We ran BCC-TI with no caching and with P-caching, and compared response times and miss rates. Imprecision limits are set to zero to enforce serializability. Fig. 1a shows average query response time for different arrival rates of update transactions.

Fig. 1. Effect of Caching on BCC-TI



The response time increases as updates become more frequent, since more and more queries are aborted due to object copies becoming stale. When caching is used, the average wait time for a read is reduced since the query can use a cached copy instead of having to wait for the next broadcast of an object. Moreover, the earlier a query completes execution, the less likely it is to abort due to its copies becoming stale. Fig. 1a shows that caching is most useful when updates are rare. The improvement in response time is about 28.5% at the lowest update arrival rates, and about 12.2% at the highest update arrival rate.

As updates become more frequent, caching becomes less beneficial, because cache entries are frequently invalidated. The reduction in response time is reflected in a decreased deadline miss rate (Fig. 1b). Fig. 1c shows the average response time of a query, given different cache sizes. As the cache size increases, more objects can be stored in the cache, and it is more likely that a query will find a cached copy of the object. Caching is most effective at the maximum cache size (when all the objects accessed by the client are cached), reducing the average query response time by up to 64% at an update arrival time of 100 Kbit time units.

5.2 Experiment 2: Quasi-TI compared to BCC-TI

BCC-TI enforces serializability, while Quasi-TI enforces quasi-consistency. Quasi-consistency can reduce the number of aborts because a query's semantics may allow it to ignore updates to previously-read data, within certain tolerable limits. Reducing aborts reduces the average response time because fewer queries have to be restarted.

Fig. 2. Effect of Update Arrival Rate on BCC-TI and Quasi-TI

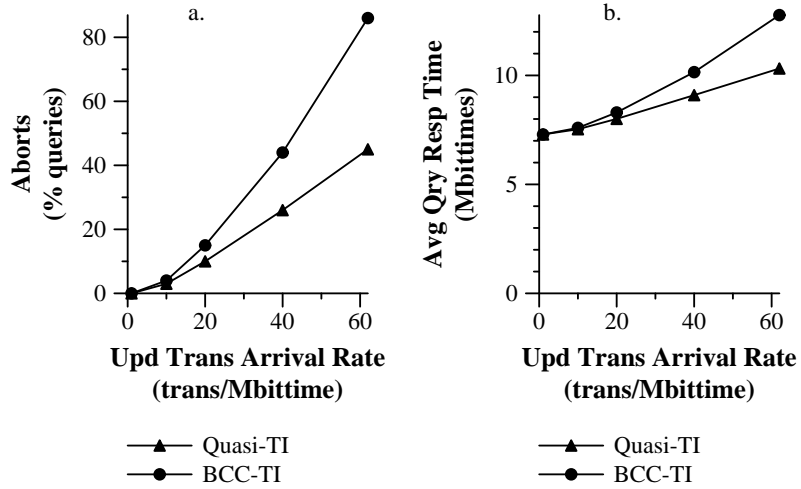
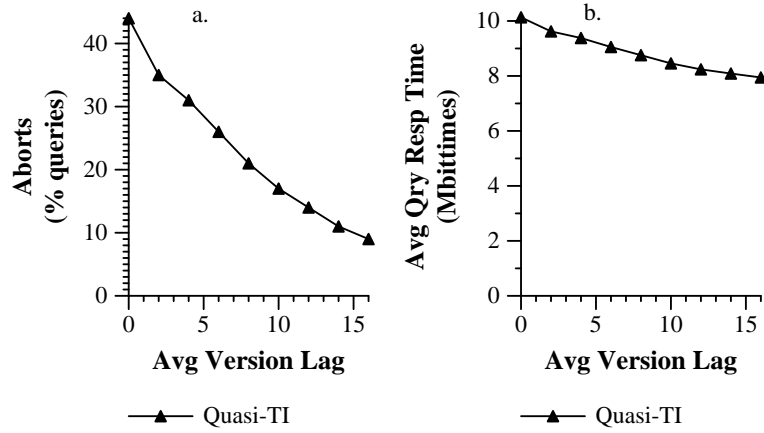


Fig. 2a shows the percentage of queries aborted as the update arrival rate is varied. As updates become more frequent, aborts in BCC-TI increase rapidly, while aborts in Quasi-TI increase less rapidly. As a result of the increased aborts, the average response time increases more rapidly for BCC-TI than for Quasi-TI (Fig. 2b). The UpdArrivTime used for this experiment is 25 Kbit time units. Quasi-TI's response time grows very slowly because queries stay within tolerable imprecision limits even when objects that they have read are updated. The reduction in the percentage of aborts by Quasi-TI is up to 41%, and the resulting improvement in response time of Quasi-TI as compared to BCC-TI is up to 19.2%. The decrease in query response time manifests itself as a reduced deadline miss rate (Fig. 2c).

The looser the consistency constraints of queries, the less likely it is that the queries will be aborted. Fig. 3a shows percentage of queries restarted as the looseness of consistency constraints is varied. The looser the constraints, the higher the values of AvgImp, AvgVLag and AvgTLag. A looseness of zero represents BCC-TI. As

constraints become looser, aborts decrease, reducing query response time, letting Quasi-TI outperform BCC-TI by up to a 21.6% reduction in response time (Fig. 3b).

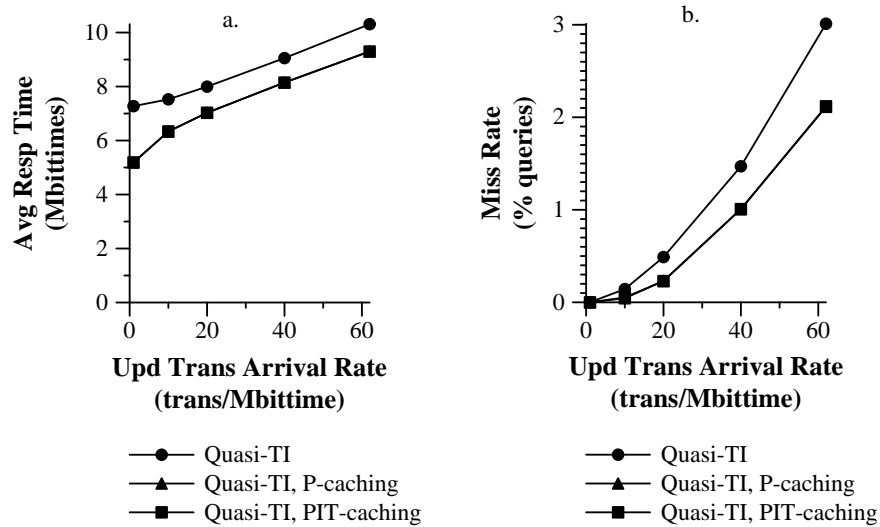
Fig. 3. Effect of Constraint Looseness on BCC-TI and Quasi-TI



5.3 Experiment 3: Caching in Quasi-TI

In this experiment, we studied caching effects on Quasi-TI. Recall that in addition to P-caching, a new scheme called PIT-caching becomes appropriate for Quasi-TI. We ran Quasi-TI without caching, and then with P-caching and PIT-caching, and compared the average query response time and deadline miss rate. Caching benefits Quasi-TI if consistency constraints are loose and there are a significant number of queries which don't care about the imprecision in object values.

Fig. 4. Performance of Caching in Quasi-TI



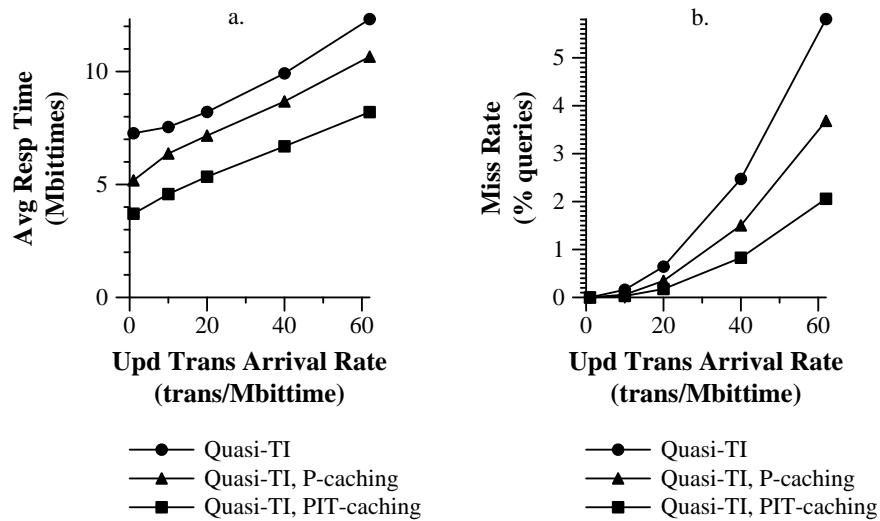
The PercNoImp used from now on is 50, and UpdArrivTime is 25. Fig. 4a shows query response time as update transaction arrival rate is increased. The difference between P-caching and PIT-caching is very small and not apparent in the graphs for response time and miss rate. P-caching reduces the response time of Quasi-TI by 28.7% (PIT-caching by 28.6%) when updates are rare. With frequent updates, the improvement in response time goes down to 9.7% for P-caching (9.8% for PIT-caching). This is because at high update rates, more cache copies become invalid, and queries have to read objects off the air. The average wait time to read an object off the air is half a broadcast cycle. The longer execution time makes an abort more likely, and an abort implies a longer response time because the query is restarted. A long response time means that the query is more likely to miss its deadline (Fig. 4b).

P-caching and PIT-caching perform similarly. PIT-caching would perform better if there were a large number of objects that are frequently accessed but have such tight consistency constraints that caching them would be useless. We conjectured that our workload did not have this property. In order to confirm this hypothesis, we modified our workload to have the above property for experiment 4.

5.4 Experiment 4: Caching in Biased Quasi-TI

In order to confirm the above hypothesis, we modified our workload so that the most-frequently-read objects had the tightest consistency constraints, that is, tolerable imprecision limits of zero. We ran Quasi-TI without caching, with P-caching and with PIT-caching, on this modified (biased) workload. Fig. 5a shows query response time as update transaction arrival rate increases. PIT-caching now outperforms P-caching by 23-28%, due to the fact that objects cached in P-caching become invalidated frequently, while PIT-caching selects objects that have loose consistency constraints and that are therefore less frequently invalidated.

Fig. 5. Performance of Caching in Biased Quasi-TI

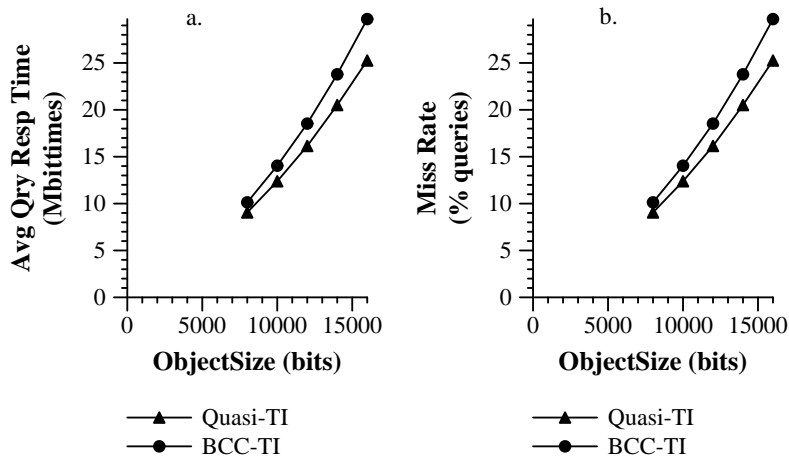


The frequent invalidation of cache entries in P-caching has two implications: [1] Queries don't often find usable cached copies. [2] Even when queries are able to use cached copies, they may be aborted due to the copy becoming invalid. The benefits of PIT-caching are tempered by the fact that the objects that it caches are not accessed frequently. The improvement in response time is reflected in the miss rate (Fig. 5b).

5.5 Experiment 5: Effect of Object Size

As object size increases, the length of the broadcast cycle increases. Therefore the average wait time for a read becomes higher, increasing the query span. A longer query span implies a higher probability of data conflict and hence a higher abort probability. Therefore, the average response time (in BCC-TI and Quasi-TI) increases with object size. The deadline miss ratio in both techniques also increases, because it is more difficult for a query to meet its deadline after an abort. BCC-TI is slightly more sensitive to object size than Quasi-TI because Quasi-TI has fewer aborts and restarts than BCC-TI. The performance of BCC-TI and Quasi-TI for different object sizes are shown in terms of average response time (Fig. 6a) and deadline miss rate (Fig. 6b). The difference in average response time of the two schemes varies from 10.6% at small object sizes to about 15% at larger object sizes.

Fig. 6. Effect of Object Size



6 Conclusions

We have presented a new CC technique for asymmetric communication environments (Quasi-TI), which allows queries to specify semantic consistency constraints and enforces these constraints. Quasi-TI can result in fewer aborts, a lower query response time and a lower deadline miss rate than BCC-TI. Through simulation, we have demonstrated the benefits of caching for BCC-TI and Quasi-TI. We have presented a new caching scheme, PIT-caching, for quasi-consistent systems. BCC-TI

benefits from P-caching when updates are rare and the cache is large. Quasi-TI outperforms BCC-TI, especially when updates are frequent and consistency constraints are loose. Caching benefits Quasi-TI when consistency constraints are loose and a significant number of queries leave their tolerable imprecision in object values unspecified. PIT-caching outperforms P-caching when many frequently-accessed objects have tight consistency constraints. We plan to extend this work to multiple broadcast disks and study broadcast-frequency-based caching. Prefetching objects into the cache is an interesting issue. We plan to run extensive tests with different workloads and study the effect of matching hot spots at server and client. Finally, we would like to explore other caching schemes.

References

1. Lee V., Son S. H., Lam K.: On the Performance of Transaction Processing in Broadcast Environments, *Int Conf on Mobile Data Access (MDA'99)*, Hong Kong, Dec 1999
2. Alonso R., Barbara D., Garcia-Molina H.: Data Caching Issues in an Information Retrieval System, *ACM Transactions on Database Systems (TODS)*, 15/3, Sep 1990
3. Pitoura E., Bhargava B.: Building Information Systems for Mobile Environments, *Proc of the 3rd Int Conf on Information and Knowledge Management*, pp 371-378, 1994
4. Acharya S., Alonso R., Franklin M., Zdonik S.: Broadcast Disks: Data Management for Asymmetric Communication Environments, *ACM SIGMOD*, 1995
5. Alonso R., Korth H.: Database Systems Issues in Nomadic Computing, *Proc of ACM SIGMOD Conf*, Washington DC, pp 388-392, 1993
6. Barbara D., Imielinski T.: Sleepers and Workaholics: Caching Strategies in Mobile Environments, *Proc of the 1994 ACM SIGMOD Conf*, pp 1-12, 1994
7. Dunham M. H., Helal A., Balakrishnan S.: A Mobile Transaction Model that Captures Both the Data and Movement Behavior, *Mobile Networks and Applications* (2), 1997
8. Imielinski T., Badrinath B. R.: Mobile Wireless Computing: Challenges in Data Management, *Communications of the ACM* 37/10, pp 18-28, 1994
9. Lee V. C. S., Lam K. Y., Tsang W. H.: Transaction Processing in Wireless Distr Real-Time Database Systems, *Proc 10th Euromicro Workshop on Real Time Systems*, Jun 1998
10. Shekar S., Liu D.: Genesis and Advanced Traveler Information Systems (ATIS): Killer Applications for Mobile Computing, *MOBIDATA Workshop*, New Jersey, 1994
11. Zdonik S., Alonso R., Franklin M., Acharya S.: Are Disks in the Air Just Pie in the Sky, *Proc of Workshop on Mobile Computing Systems and Applications*, California, 1994
12. Acharya S., Franklin M., Zdonik S.: Disseminating Updates on Broadcast Disks, *Proc of the 22nd VLDB Conf*, Bombay, India, 1996
13. Herman G., Gopel G., Lee K. C., Weinrib A.: The Datacycle Architecture for Very High Throughput Database Systems, *Proc of ACM SIGMOD Conf*, pp 97-103, 1987
14. Imielinski T., Viswanathan S., Badrinath B. R.: Energy Efficient Indexing on Air, *Proc of ACM SIGMOD Conf*, May 1994
15. Shanmugasundaram J., Nithrakashyap A., Sivasankaran R., Ramamritham K.: Efficient Concurrency Control for Broadcast Environments, *ACM SIGMOD*, 1999
16. Lee J., Son S. H.: Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems, *Proc 14th IEEE Real-Time Systems Symposium*, pp 66-75, 1993
17. Pitoura E., Chrysanthos P. K.: Scalable Processing of Read-Only Transactions in Broadcast Push, *Proc 19th IEEE Int Conf on Distributed Computing Systems*, 1999