

A Real-Time Database Testbed and Performance Evaluation

Kyoung-Don Kang, Phillip H. Sin, and Jisu Oh
Department of Computer Science
State University of New York at Binghamton
{kang,joh}@cs.binghamton.edu,
bj94041@binghamton.edu

Sang H. Son
Department of Computer Science
University of Virginia
son@cs.virginia.edu

Abstract

A lot of real-time database (RTDB) research has been done to process transactions in a timely fashion using fresh data reflecting the current real world status. However, most existing RTDB work is based on simulations. Due to the absence of a publicly available RTDB testbed, it is very hard to evaluate real-time data management techniques in a realistic environment. To address the problem, we design and develop an initial version of a RTDB testbed, called RTDB2 (Real-Time Database Benchmark), atop an open source database [5]. We develop soft real-time database workloads that model online stock trades, providing several knobs to specify workloads for RTDB performance evaluation. In addition, we develop a QoS management scheme in RTDB2 to detect overload and reduce workloads, via admission control and adaptive temporal data updates, under overload. From the extensive experiments using the stock trading workloads developed in RTDB2, we observe that adaptive updates can considerably improve the transaction timeliness. We also observe that admission control can only enhance the timeliness under severe overload, possibly causing underutilization problems for moderate workloads.

1 Introduction

In a number of real-time applications, e.g., stock trading, agile manufacturing, and traffic control, real-time databases (RTDBs) are required to process transactions in a timely fashion using a large number of temporal data, e.g., current stock prices or traffic sensor data, representing the real world status. RTDBs can relieve the difficulty of developing data-intensive real-time applications by supporting the logical and temporal consistency of data via transactions. Also, in these applications, RTDBs can significantly outperform ex-

isting non-real-time databases (non-RTDBs) unaware of timing and data freshness, i.e., temporal consistency, requirements [15, 16].

RTDBs have been studied for more than a decade producing key results; however, most existing work on RTDBs is based on simulations [16], which have limitations in modeling realistic workloads and real database system dynamics. Very little prior work such as [1, 10] has been done to evaluate real-time data management techniques in real database systems. There are several commercial RTDB products [11, 18, 14], but they are not open to the public. They do not apply the latest RTDB technology too. The lack of an open RTDB benchmark is an obstacle for more active RTDB research and application development in a realistic environment. To shed light on the problem, we design and develop a RTDB testbed, called RTDB2 (Real-Time Database Benchmark) on top of Berkeley DB [5]. To our best knowledge, RTDB2 is the first open source RTDB testbed.

RTDB2 models online stock trading, which is a data-intensive *soft real-time* application. A stock brokerage service consists of a large number of clients and their transactions for online quotes and trades needed to track, evaluate, and manage investments. Databases are a key component of such services as stock brokerage, because they support the ACID (atomicity, consistency, isolation, and durability) properties of transactions essential for trades [7]. Real-time data services are required to process transactions within the specified delay bound, e.g., 3s, and maintain the freshness of stock prices in addition to supporting the ACID properties. Note that we do not consider individual transaction deadlines, because most online transactions are not associated with separate deadlines. Instead, it is desired for a RTDB to process as many trade transactions as possible within a systemwide delay bound to avoid losing a majority of clients due to excessive service delays [4, 21].

Although there are existing non-RTDB testbeds such as the TPC (Transaction Processing Performance Council) benchmarks [19], they mainly focus on the average response time and throughput (i.e., transactions/minute), which are not RTDB performance metrics [17, 16]. They do not consider data freshness constraints and timing requirements expressed by deadlines or a response time bound. Due to the complexity, it is very hard to modify non-RTDB benchmarks for RTDB performance evaluation. Further, most of them are proprietary. For these reasons, we develop a novel RTDB testbed to directly consider timing and freshness requirements by modeling stock trade workloads. RTDB2 provides several knobs by which one can control transaction and update workloads applied to RTDB2 for timeliness and freshness evaluation under different load conditions. Overall, RTDB2 is a starting point to develop a realistic RTDB testbed.

We also develop a RTDB QoS management scheme in RTDB2 to detect overload and adjust the workload to process as many trade transactions as possible within the desired delay bound for real-time data services. To detect overload and determine the required workload adjustment, we compute the degree of timing constraint violation based on the difference between the actual response time and the desired delay bound. RTDB2 applies admission control to avoid database thrashing under overload [10, 12] considering the degree of timing constraint violations rather than relying on (estimated) transaction execution times [10, 9, 2], which are hard to predict in databases involving transaction aborts and restarts [15].

Frequent temporal data updates in RTDBs can consume a lot of system resources [1, 9]. However, transactions do not always access all temporal data in a uniform manner. Hot data, e.g., popular stock prices, can be accessed more often than cold data. To improve the transaction timeliness under overload, cold data can be updated less frequently as long as the freshness of temporal data accessed by user transactions is not affected by more than the specified bound [9]. Specifically, we reduce the overhead of the adaptive update policy [9] to efficiently adjust update workloads when overloaded.

For performance evaluation, we have undertaken extensive experiments using the developed stock trading workloads in RTDB2 to compare the performance of admission control and adaptive temporal data updates to a baseline approach, which simply accepts all incoming tasks and updates every temporal data without relaxing the freshness constraints, similar to most existing database systems with no QoS management. Our adaptive update scheme can considerably improve the transaction timeliness, while supporting the desired

freshness requirements, compared to the baseline. On the other hand, we observe that admission is only effective under severe overload. Admission control generally achieves the best *average response time*; however, for moderate workloads, it achieves a lower *success rate*, i.e., the number of timely transactions that finish with the delay bound per unit time, than the baseline and adaptive update scheme. These results experimentally verify that an average performance metrics is inappropriate for RTDB performance evaluation [17]. Even though admission control is known to be effective for overload protection in real-time systems [6], it may not be directly applicable to RTDB performance management unless transaction execution times, arrival patterns, and data access patterns are known in advance. Thus, from these experiments in a real system, we find that database specific nature should be considered for RTDB QoS management.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. Our RTDB architecture, QoS management scheme, and database schema and workloads modeling stock trading are discussed in Section 3. Performance evaluation results are described in Section 4. Finally, Section 5 concludes the paper and discusses future work.

2 Related Work

Real-time transaction processing is not fast transaction processing [17]. Simplistic fast processing could actually reduce the transaction timeliness. A significant amount of research has been done in real-time transaction scheduling and concurrency control to support the transaction timeliness in RTDBs [3, 16]. However, most existing work is based on simulations. Therefore, realistic experiments are required to compare and verify existing core techniques for RTDB performance management.

STRIP (STanford Real-time Information Processor) [1] is a real-time database system originally developed for research and commercialized later. EagleSpeed [11], Polyhedra [14], and TimesTen [18] are developed for real-time transaction processing, but they are proprietary. BeeHive [10] is a recent effort for developing a RTDB testbed. It supports advanced real-time transaction scheduling based on the data deadline and forced wait concepts [23]. However, BeeHive is not publicly available.

APPROXIMATE [22] is an approximate real-time query processing framework. The accuracy of a query answer monotonically increases as a query is executed longer, while an imprecise answer can be returned to meet the transaction deadline. Analogously, database

sampling is used to process real-time queries using a subset of data in the database, if necessary, to meet their deadlines [13]. Adelberg et. al. [1] observe that there is a trade-off between transaction timeliness and data freshness. If temporal data updates receive a higher priority the data freshness can be improved, while the transaction timeliness can be reduced and vice versa. They propose several transaction scheduling algorithms to balance the conflicting freshness and timeliness requirements in RTDBs.

Several recent work [9, 2, 8], via simulations, has studied QoS management issues for systematic performance trade-offs in RTDBs. They aim to support the desired transaction timeliness and data freshness for dynamic workloads by feedback control and QoS management techniques such as adaptive temporal data updates, imprecise transaction processing, service differentiation, and admission control. In the future, key techniques for RTDB QoS management can be implemented and evaluated in RTDB2. As an initial effort, in this paper, we apply admission control and adaptive temporal data updates to improve the timeliness of soft real-time transactions under overload.

3 Real-Time Database Testbed Design and QoS Management

In this section, RTDB2 architecture, RTDB2 schema and transactions, and overload detection and management schemes are discussed.

3.1 RTDB2 Architecture

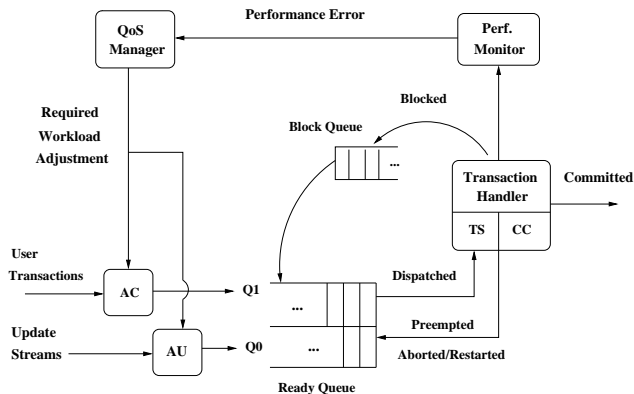


Figure 1. RTDB2 Structure

Figure 1 shows the overall structure of RTDB2 consisted of the admission control (AC), adaptive update (AU), transaction scheduling (TS), concurrency control (CC), performance monitoring, and QoS management

components. RTDB2 users can configure to turn on or off these components individually for performance evaluation purposes. In RTDB2, periodic temporal data updates receive a higher priority than user transactions to ensure the required data freshness, similar to [15, 9, 23]. The transaction scheduler applies the fixed priority between user and update transactions in Q_1 and Q_2 in the figure, while scheduling transaction in a FCFS manner in each queue. For concurrency control, we currently apply 2PL provided by Berkeley DB. A transaction can be blocked, aborted, and restarted due to data conflicts. Once blocked, it waits for the conflicting transaction(s) to finish in the block queue. One can replace the FCFS and 2PL with real-time transaction scheduling and concurrency control mechanisms. This is reserved for our future work. The performance monitor detects overload by periodically computing the performance error, i.e., the difference between the measured response time and the desired delay bound. In proportion to the error, the QoS manager shown in Figure 1 computes the required workload adjustment to be used for admission control and adaptive updates under overload.

RTDB2 follows the client-server model. To receive RTDB services, a client thread first needs to send the server a TCP connection request. It suspends until the server accepts the connection request and allocates a server thread. When the connection is established, the client thread sends a transaction or query (i.e., read-only transaction) processing request and suspends until the corresponding server thread finishes processing the transaction and returns the result. After receiving the result, the client thread waits for a think time uniformly selected in a configurable range, e.g., [0.5s, 1s], before issuing another request, similar to [20, 4, 21]. For performance evaluation purposes, one can specify the number of maximum client processes and threads. As more client processes and threads are created, the workload generally increases due to more data/resource contention caused by concurrent transactions. In this way, we can model realistic workloads in which hundreds or thousands of concurrent transactions compete for data and resources.

Update transactions in Figure 1 periodically update the 3,000 stock data in the RTDB2 server by pulling and processing the corresponding stock prices from Yahoo! Finance [24]. (Most free quote services including Yahoo do not provide push services where the data provider disseminates data.) Currently, there are 100 update threads in a RTDB2 server. Each thread periodically wakes up to pull predefined 30 stock prices from Yahoo! Finance. One can specify the update periods of temporal data, i.e., stock prices in this paper,

to determine the freshness requirements of temporal data [15, 9] and corresponding update workloads. By assigning short update periods, one can increase the update workload and vice versa. Also, one can allocate more update threads to further increase the number of temporal data and update workloads. Thus, the current version of RTDB2 provides three knobs for workload specification: (1) the number of client threads and range of think time, (2) the number of temporal data and update threads in the RTDB2 server, and (3) the update periods of temporal data.

3.2 Database Schema and Transactions

The current version of RTDB2 consists of seven tables. In this paper, we only consider data access to the four tables: Stocks, Quotes, QuoteHist (Quote History), and Portfolios. The small number of tables does not limit the capability of RTDB performance evaluation, since complicated tables such as Quotes and Portfolios should be updated very frequently, while Stocks and QuoteHist tables are necessary to make trade decisions. TPC-C [20], which is a well known benchmark for online transaction processing modeling data warehousing, also supports five types of transactions on fewer than 10 tables, similar to RTDB2. Compared to TPC-C, our testbed more closely models RTDB transactions involving significant periodic temporal data updates and timing constraints as discussed before. Specifically, RTDB2 consists of the following tables and data attributes:

- The **Stocks** table consists of the stock symbol, full name, and ID for each company.
 - In the **Quotes** table, the company ID can be used to uniquely identify a company’s current quote information. This table’s columns corresponds to the Yahoo! Finance quotes’ attributes. It is most complicated in RTDB2; it consists of 17 columns including the current price, trade time, low and high price of the day, percentage of price change, bidding price, asking price, trade volume, and market capitalization for each company. For more details, refer to [24].
 - The **QuoteHist** table keeps track of historical values for each company. It has the same structure as the Quotes table. A new quote data is appended to QuoteHist for every update. Thus, a user can query stock price trends, which can be important for making investment decisions. Querying and updating QuoteHist can create substantial I/O operations as the the number of history data in the table grows, affecting RTDB2 performance.
 - The **Portfolios** table has the account ID, company ID, and own flag to list all stock purchases and sale orders for each client. One account ID can correspond to multiple rows, i.e., portfolios. In the current version of RTDB2, each client is associated with 50 – 100 portfolios.
 - The **Accounts** table maintains login information of each client consisted of the account ID, user name, and password attributes.
 - The **Currencies** table lists 92 country’s currencies and their exchange rates for US dollar. It consists of the country name, currency name, and exchange rate attributes.
 - The **Personal** table is composed of the account ID, last name, first name, address, address2, city, state, country, phone number, and email address attributes.
- In addition to temporal data updates, RTDB2 supports four types of user transactions that can read, write, insert, and delete data to model stock trades.
- **VIEW-STOCK**: A client can request, via this transaction, to select a specified set of companies’ information and their associated stock quotes.
 - **VIEW-PORTFOLIO**: A client can request to select its portfolios and see the associated stock quotes.
 - **PURCHASE**: A client can place a purchase order for selected stocks.
 - **SELL**: A client can require program trading, in which the server automatically trades the specified stock items in his/her portfolios when the values of the stocks in a portfolios change by more than a specified threshold.
- After establishing a connection with the server, a client thread can request one of these transactions at a time, possibly requesting a sequence of transactions dispersed by think times before closing the connection. Clearly, our schema and transactions are not the only way of designing and developing a RTDB testbed. Other schema and transactions for different applications, e.g., traffic control, can also be implemented in RTDB2.

3.3 QoS Management

In this section, the overload detection, admission control, and adaptive update schemes supported by RTDB2 are described.

3.3.1 Overload Detection

It is desired to limit the response time of a RTDB to be below a certain threshold to maintain clients [4]. However, a database system can be overloaded if many users transactions are executed concurrently. As a result, computational resources such as CPU cycles and memory space can be exhausted. Moreover, many transactions can be blocked or aborted and restarted due to data contention.

To detect overload, we define the degree of timing constraint violations. Let t_s be the desired delay bound and t_m be the response time measured periodically. In this paper, we measure the performance at every 30s to ensure that we have an enough number of transactions committed within the sampling period. (Up to approximately 1800 transactions are processed within 30s in our experiments discussed in Section 4.) If $t_m > t_s$, RTDB2 is considered overloaded and the degree of overload at the k^{th} measurement period is:

$$\delta(k) = (t_m(k) - t_s)/t_s \quad (1)$$

For instance, $\delta(k) = 0.2$ when $t_m = 3.6s$ and $t_s = 3s$.

In reality, workloads may vary from time to time. Accordingly, the response time may vary from a measurement to another. If admission control and adaptive update schemes are applied based on instantaneous δ values, RTDB performance may significantly fluctuate. To address the problem, we take an exponential moving average of δ over several measurement periods. The smoothed value $\delta_s(k)$ at the k^{th} measurement period is:

$$\delta_s(k) = \alpha \cdot \delta(k) + (1 - \alpha) \cdot \delta_s(k - 1) \quad (2)$$

where $0 \leq \alpha \leq 1$ is a tunable parameter. If $\alpha = 1$ in Eq 2, $\delta_s(k)$ only takes the current $\delta(k)$ value into account to compute $\delta_s(k)$, while it considers a wider horizon as a smaller value of α is chosen. In this paper, we set $\alpha = 0.6$ to ensure that $\delta(k - 5)$, i.e., the δ value measured five sampling periods ago, have a weight less than 0.01 in computing $\delta_s(k)$.

3.3.2 Admission Control

To avoid database thrashing due to severe data/resource contention, RTDB2 applies admission control to incoming transactions when $\delta_s(k) > 0$. For example, if $\delta_s(k) = 0.2$, our admission control scheme tries to reduce the number of concurrent transactions by 20% to support the desired response time bound. However, the database system may not be able to immediately kill transactions for database consistency reasons. Thus, in this example, the database system has to wait for 20% of the transactions currently in the

system to finish, decrementing $\delta_s(k)$ when one transaction finishes within the k^{th} measurement period. A new transaction can be admitted when $\delta_s(k)$ becomes negative within the period, while $\delta_s(k + 1)$ is computed at the beginning of the $(k + 1)^{th}$ period. BeeHive is a RTDB system supporting admission control; however, transaction execution times should be determined offline for admission control [10]. This approach is only applicable to a limited set of real-time data services in which transactions and their arrival/data access patterns are predetermined or highly predictable.

3.3.3 Adaptive Update Policy

To reduce update workloads under overload, we adopt the notion of *flexible validity intervals* [9] that can gracefully relax absolute validity intervals [15] used to maintain the data temporal consistency in RTDBs. When data d_i 's absolute validity interval is $avi[i]$, its update period $p[i] = 0.5avi[i]$ to maintain the freshness of the data [15]. The adaptive update policy [9] based on the flexible validity interval (*fvi*) concept computes the access update ratio $AUR[i]$ for each temporal data d_i in the RTDB according to its update frequency, i.e., $1/p[i]$, and the measured access frequency:

$$AUR[i] = \frac{Access\ Frequency[i]}{Update\ Frequency[i]} \quad (3)$$

If $AUR[i] \geq 1$, d_i is considered hot; otherwise, it is considered cold. When the system is overloaded, a fraction of cold data can be updated less frequently by increasing their update periods within a pre-specified bound. Let β indicate the update period relaxation bound. For each temporal data d_i , $fvi[i] = avi[i]$ initially. Given β , a RTDB can increase the update period of an arbitrary cold data d_i at runtime as long as the following condition is met:

$$avi[i] \leq fvi[i] \leq \beta \cdot avi[i] \quad (4)$$

To maintain d_i 's flexible temporal validity, $p[i]$ is always equal to $0.5fvi[i]$ in RTDB2.

The coldest data with the lowest AUR value is degraded first in [9]; however, this requires sorting the AUR's. The complexity of sorting is $O(n \log n)$ for n temporal data. To reduce the overhead, we check each cold data whether its update period can be further increased by a pre-specified value, e.g., 10%, without violating the condition described in Eq. 4. Thus, the overhead is reduced to $O(n)$. This freshness degradation is repeatedly applied to up to $\delta_s(k) \cdot n$ cold data at each freshness adaptation period, which is equal to the overload detection period described before. Our

overload detection, admission control, and adaptive update schemes can be further improved for more efficient overload management. For example, control theoretic approaches [9, 2, 8] can be applied to support the desired response time even when the system is in a transient status. A thorough investigation is reserved for future work.

4 Performance Evaluation

In this section, we evaluate the performance of RTDB2 for an increasing number of client threads. For a RTDB2 server, we use a Dell laptop with the 1.66 GHz dual core CPU and 1 GB memory, which runs Linux with the 2.6.15 kernel. We use two Dell desktop PCs to create up to 1,800 client threads. One PC has the 3 GHz CPU and 2GB memory, while the other one has the same CPU and 4 GB memory.

The client and sever machines are connected via a 100 Mbps Ethernet switch. Each client machine generates between 300 to 900 client threads. Therefore, we generate 600–1800 client threads for performance evaluation. A client thread can issue one of the four user transactions described in Section 3. More specifically, 60% of client requests are `View-STOCK` and other 40% transactions are uniformly selected among the other three types of transactions. The number of data accesses in one transaction varies between 50 and 100. The think time uniformly distributed in $[0.3s, 0.5s]$. There is a single RTDB2 server process in our experiments. The size of the TCP connection queue in the server is 2000 and the server can run up to 350 concurrent threads to process user transactions.

In RTDB2, `Quotes` table maintains the 3000 stock data. For arbitrary temporal data $d[i]$ in the database, its flexible validity interval $fvi[i] = 1s$ and update period $p[i] = 0.5s$ initially to support the freshness. We set the update period relaxation bound $\beta = 2$. Hence, the update period of a temporal data can be increased up to 2s in this paper. As described in Section 3, RTDB2 maintains the update period $p[i] = 0.5fvi[i]$ to support the data temporal consistency.

Given the update and user transaction workload settings, we compare the performance of (1) *Berkeley DB* (BASE), (2) *Admission Control* (AC), and (3) *Adaptive Update Policy* (AUP) for an increasing number of client threads. In BASE, we apply neither admission control nor adaptive update policy. Thus, we evaluate the performance of Berkley DB underlying RTDB2. We observe whether or not AC and AUP described in Section 3.3 can improve the performance. One experiment lasts for 15 minutes. Each performance data presented in this paper is the average of 5 runs. 90% confidence in-

tervals are also derived and plotted as vertical bars in the graphs.

4.1 Success Rate

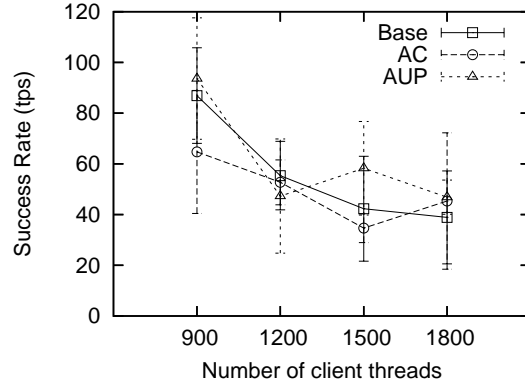


Figure 2. Success Rate

In this paper, we use 3s as the desired delay bound. Figure 2 shows the success rate, i.e., the number of timely user transactions per second (ttps) that complete within the desired 3s response time bound. Generally, the success rate of AUP is the highest among the three approaches. For 900 client threads, AUP achieves 93.62 ttps and BASE supports 86.92 ttps. The success rate of BASE quickly drops as the number of client threads increases. For 1800 client threads, its success rate is only 38.87 ttps due to severe overloads. A drawback of AUP and BASE is the relatively large confidence intervals shown in the figure. In our experiments, approximately 20% to 30% transactions show substantially long response time. For example, the longest response time of BASE is 89.25s in a sampling period when the number of client threads is 1200. This is because some transactions access a larger number of data than the others incurring many data/resource contention.

This is because AUP depends on potentially time-varying data access patterns and the strict upper bound for freshness degradation expressed by β .

AC shows the lowest success rate when the number of client threads ≤ 1500 . It implies that applying admission control to moderate workloads can impair the success rate due to unnecessary transaction rejections. Under overload, however, AC performs well. For 1800 client threads, the success rate of AC is 45.34 ttps, which is close to the success rate of AUP. Admission control becomes too pessimistic in our experiments, since a subset of transactions suffering long response times significantly affect the average response

time statistics taken at a performance measurement period. As a result, too many transactions are dropped. In BeeHive [10], admission control—based on execution times analyzed offline—is observed to be effective for mainly overload conditions, similar to our results. To avoid the low success rate due to underutilization, admission control needs to be applied only under severe overload with significant data/resource contention. To further enhance the success rate, AUP can be applied first for overload management, while applying AC under severe overload conditions. A thorough investigation of more sophisticated admission control and its interactions with adaptive updates is reserved for future work.

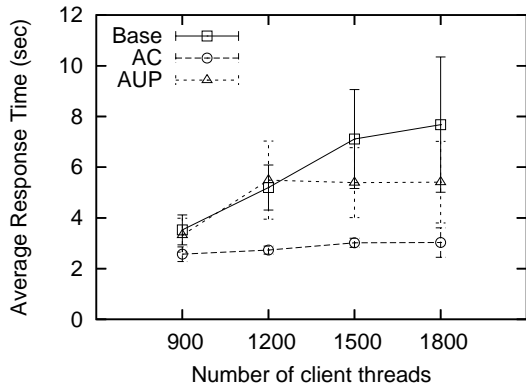


Figure 3. Average Response Time

Figure 3 shows that the average response time—a performance metric commonly used in non-RTDB benchmarks—is misleading. In the figure, AC, which generally achieves the lowest success rate in Figure 2, shows the best average response time. Thus, not the average response time but a real-time performance metric that can measure the timeliness of individual transactions, e.g., the success rate, should be considered for RTDB performance management and evaluation.

4.2 Freshness

Figure 4, shows the average degree of update period extension $P_{ext} = \frac{1}{N} \sum_{i=1}^N \frac{p[i]}{p[i]_{init}}$ where N is the total number of the temporal data in the database, $p[i]$ is the potentially extended update period of temporal data $d[i]$, and $p[i]_{init}$ is the initial update period of $d[i]$. When $P_{ext} = 2$, the update period of every temporal data is doubled, i.e., $p[i] = 1s$ for arbitrary data $d[i]$ in RTDB2. Since the update period relaxation bound $\beta = 2$, the result ranges in $[1.0, 2.0]$ as shown in the figure. Thus, the freshness requirement considered in this

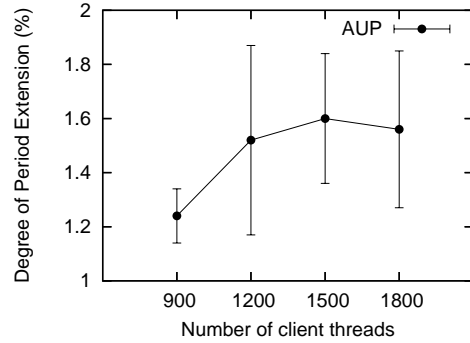


Figure 4. Degree of Update Period Extension

paper is met. As the number of client threads increases, AUP can extend more update periods to support the desired delay bound for more user transactions. As a result, P_{ext} generally increases for the increasing number of the client threads, while satisfying the freshness requirements.

5 Conclusions and Future Work

Most existing RTDB research is based on simulations using synthetic workloads. There is no open RTDB testbed. Thus, evaluating real-time data management techniques in a realistic environment is very difficult. To address this problem, we develop an initial version of a RTDB testbed called RTDB2 to model stock trading. Timing and data freshness constraints are considered throughout the design, development, and evaluation of RTDB2. We also develop overload detection, admission control, and adaptive update schemes to enhance the transaction timeliness under overload. Via extensive experiments using the developed stock trade workloads in a real system, we observe that adaptive updates can considerably improve the transaction timeliness compared to the underlying database, which does not have overload detection and load shedding schemes. In contrast, we find that admission control needs to be applied under severe overload conditions only. In the future, we will further enhance our RTDB testbed. We plan to evaluate key real-time transaction scheduling and concurrency control algorithms. Further, we will investigate new techniques for RTDB QoS management.

References

- [1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD*, 1995.

- [2] M. Amirijoo, N. Chaufette, J. Hansson, S. H. Son, and S. Gunnarsson. Generalized Performance Management of Multi-Class Real-Time Imprecise Data Services. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 38–49, 2005.
- [3] A. Bestavros, K. J. Lin, and S. H. S. (editors). *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, 1997.
- [4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-Perceived Quality into Web Server Design. In *9th International World Wide Web Conference*, 2000.
- [5] Oracle Berkeley DB Product Family, High Performance, Embeddable Database Engines. Available at <http://www.oracle.com/database/berkeleydb/index.html>.
- [6] G. C. Buttazzo and G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
- [7] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 4th edition, 2003.
- [8] K. D. Kang, S. H. Son, and J. A. Stankovic. Differentiated Real-Time Data Services for E-Commerce Applications. *Electronic Commerce Research*, 3(1-2), January/April 2003. Combined Special Issue: Business Process Integration and E-Commerce Infrastructure.
- [9] K. D. Kang, S. H. Son, and J. A. Stankovic. Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1200–1216, 2004.
- [10] S. Kim, S. H. Son, and J. A. Stankovic. Performance Evaluation on a Real-Time Database. In *IEEE Real-Time Technology and Applications Symposium*, 2002.
- [11] Lockheed Martin. EagleSpeed Real-Time Database Manager.
- [12] A. Möenkeberg and G. Weikum. Conflict-Driven Load Control for the Avoidance of Data-Contention Thrashing. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 632–639, 1991.
- [13] G. Ozsoyoglu, S. Guruswamy, K. Du, and W.-C. Hou. Time-Constrained Query Processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, pages 865–884, Dec 1995.
- [14] Polyhedra Plc. Polyhedra White Papers, 2002.
- [15] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.
- [16] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-Time Databases and Data Services. In *Real-Time Systems*, volume 28, Nov.–Dec. 2004.
- [17] J. Stankovic, S. H. Son, and J. Hansson. Misconceptions About Real-Time Databases. *IEEE Computer*, 32(6):29–36, June 1999.
- [18] The TimesTen Team. In-Memory Data Management for Consumer Transactions The TimesTen Approach. In *ACM SIGMOD*, 1999.
- [19] Transaction processing performance council. <http://www.tpc.org/>.
- [20] TPC-C – OLTP, Transaction Processing Performance Council. Available at <http://www.tpc.org/>.
- [21] U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of E-Commerce Traffic. *Electronic Commerce Research*, 3(1-2), 2003.
- [22] S. Vrbsky. *APPROXIMATE: A Query Processor that Produces Monotonically Improving Approximate Answers*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.
- [23] M. Xiong, K. Ramamritham, J. A. Stankovic, D. Towsley, and R. Sivasankaran. Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1155–1166, September/October 2002.
- [24] Yahoo! Finance. Available at <http://finance.yahoo.com/>.