Chronos: Feedback Control of a Real Database System Performance*

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

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

Abstract

It is challenging to process transactions in a timely fashion using fresh data, e.g., current stock prices, since database workloads may considerably vary due to dynamic data/resource contention. Further, transaction timeliness and data freshness requirements may compete for system resources. In this paper, we propose a novel feedback control model to support the desired data service delay by managing the size of the ready queue, which indicates the amount of the backlog in the database. We also propose a new self-adaptive update policy to adapt the freshness of cold data in a differentiated manner based on temporal data access and update patterns. Unlike most existing work on feedback control of real-time database (RTDB) performance, we actually implement and evaluate feedback control and database workload adaptation techniques in a real database testbed modeling stock trades. For performance evaluation, we undertake experiments in the testbed, which consists of thousands of client threads concurrently requesting database services for stock quotes, trades, and portfolio updates in a bursty manner. In these experiments, our database system supports the desired response time bound and data freshness, while processing a significantly larger number of transactions in time compared to the tested baselines.

1 Introduction

Real-time databases (RTDBs) aim to process transactions in a timely manner using fresh temporal data, e.g., current stock prices or traffic sensor data, representing the real world status. RTDBs can reduce the difficulty of developing data-intensive real-time applications such as stock trading, agile manufacturing, and traffic control, by supporting the logical and temporal consistency of data via transactions. Existing non-real-time databases are unaware of timing and data freshness (i.e., data temporal consistency) requirements, showing poor performance in these applications [18, 19].

It is challenging to provide real-time data services, since database workloads may dynamically vary due to data/resource contention. Also, transaction timeliness and data freshness may pose conflicting requirements: If user transactions are given a higher priority than temporal data updates, the transaction timeliness can be improved at the cost of the potential freshness reduction [1]. In contrast, the transaction timeliness can be reduced, if temporal data updates always have a higher priority than user transactions. Feedback control has recently been applied to manage RTDB performance in the presence of dynamic workloads [2, 8, 13], producing promising initial results. However, the existing work on feedback control of RTDB performance is based on simulations. Generally, most existing RTDB work is based on simulations [19]. Hence, there are limitations in modeling real system behaviors. Very little prior work such as [1, 9, 14] has evaluated real-time data management techniques in a real database system.

To address the problem, we have designed and implemented a soft real-time database testbed, called Chronos [7], in which thousands of clients can send the database server requests for stock quotes, trades, and user portfolio updates. We have found considerable differences between simulation-based results and experimental results in a real database system [7]. The Chronos server processes data service requests from clients, while periodically pulling and updating 3,000 stock prices from Yahoo! Finance [25][1]. Chronos

*This work was supported, in part, by NSF grants CNS-0614771 and CNS-0614886.

[1]We consider a pull model, because free-of-charge stock quote services such as Yahoo! Finance [25] do not usually provide push services, in which the data source multicasts data to the subscribers.

directly measures the achieved transaction timeliness and data freshness unlike existing non-RTDB benchmarks such as TPC (Transaction Processing Performance Council) benchmarks [21]. In this paper, we extend Chronos to generate bursty workloads. We also develop novel approaches for feedback control of RTDB performance and adaptive temporal data updates. We implement and evaluate these approaches in the extended Chronos testbed. To our knowledge, this work is the first to design, implement, and evaluate RTDB feedback control and adaptive update schemes in a real database testbed.

Although most online transactions are not associated with individual deadlines, excessive service delays may cause many clients to leave [22]. Thus, a database needs to control the response time. To support the desired service delay bound such as 2s, we develop a novel database feedback control model managing the size of the ready queue for data service requests. Generally, the size of the ready queue increases as the load increases. In this case, transaction processing is slower than transaction arrivals due to data/resource contention inside the database. On the other hand, the ready queue size decreases when the database is underutilized. Based on this observation, we model the relation between the ready queue length and response time. Note that the control model developed in this paper is different from our previous work [13] in that the feedback control model of [13] aims to support the desired CPU utilization, while the feedback control scheme of this paper intends to support the desired delay bound for real-time data services.

In addition, we develop a new self-adaptive update policy [8] to efficiently manage the freshness of temporal data. Our approach balances update and user transaction workloads considering user data needs and the current service delay by differentiating the freshness among temporal data. This approach can significantly improve the transaction timeliness under overload compared to the existing approach for adaptive temporal data updates [8] and the other approaches [19], in which a temporal data in a RTDB is updated at a fixed rate regardless of data access patterns. When the system is severely overloaded, admission control is also applied to incoming transactions. We apply admission control after freshness adaptation to improve the success ratio, i.e., the fraction of the submitted transactions finishing within the desired delay bound for data services.

For performance evaluation, we create thousands of client threads, which send data service requests to the Chronos server. As soon as a client thread finishes sending a data service request to the database server, it starts to measure the response time, while waiting for the transaction or query processing result from the server. After receiving the service, a client waits for an inter-request time that exists between most trade or quote requests. In this paper, we fix the number of the client threads and shorten the range of the inter-request time in the middle of an experiment to randomly create additional workloads ranging between 25% and 100%. In the stock market, this may happen especially when the market status is volatile. Given the bursty workloads, our approach can support the desired response time and data freshness, while significantly improving the success ratio compared to the tested baselines. Overall, our performance analysis shows the feasibility of feedback control in a real database system.

The remainder of this paper is organized as follows. Section 2 describes the Chronos architecture and feedback control system. It also gives a detailed description of the adaptive update policy. Our control modeling and tuning process is discussed in Section 3. Performance evaluation results are presented in Section 4. Related work is discussed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 RTDB Architecture and Feedback Control Overview

In this section, the architecture of Chronos and feedback control procedure are discussed. Moreover, freshness adaptation and admission control in the feedback loop are discussed. Figure 1 shows the architecture of

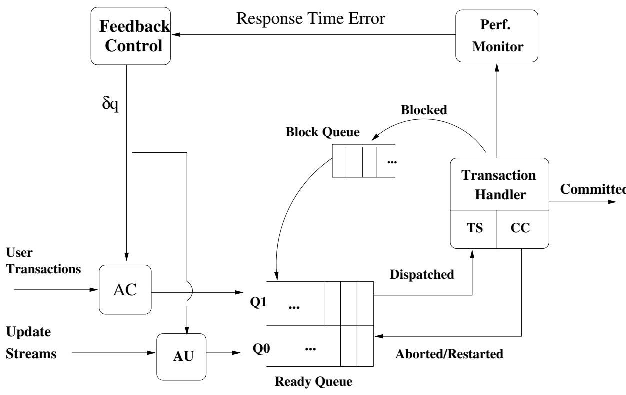

Figure 1. Chronos Architecture

Chronos that consists of the feedback control, adaptive update (AU), admission control (AC), transaction scheduling (TS), concurrency control (CC), and performance monitoring components. The performance monitor computes the response time error, i.e., the difference between the desired response time bound and

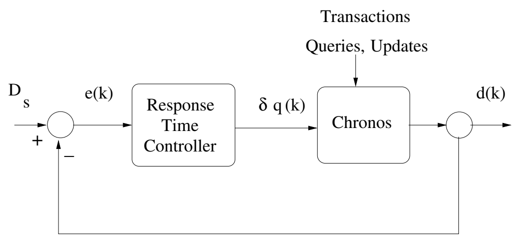

Figure 2. Response Time Control Loop

the response time measured at every sampling period. Based on the error, the feedback controller computes the required ready queue size adjustment δq. If the measured delay is longer than the desired bound, the ready queue size needs to be reduced and vice versa. According to δq, the AU and AC modules adjust workloads, if necessary, to meet the desired delay bound. One can configure Chronos to selectively turn on or off these components for performance evaluation purposes.

To support the data freshness, periodic temporal data updates scheduled in Q_0 in Figure 1 receive a higher priority than user transactions in Q_1, similar to [18, 8, 24]. Transactions in each queue is scheduled in a FCFS manner. As a user transaction arrives, it is required to finish by the time equal to the sum of the current time and relative deadline, i.e., the desired service delay such as 2s. For concurrency control, we currently apply 2PL (two phase locking) provided by Berkeley DB [3] underlying Chronos. A transaction can be blocked, aborted, and restarted due to data conflicts. Once blocked, it waits in the block queue for the conflicting transaction(s) to finish. It is reinserted into the ready queue when the data conflict is resolved.

FCFS and 2PL are the most commonly used scheduling and concurrency control algorithms in commercial databases. The theme of this work is to show that feedback control can support the desired delay bound in a database system without special instrumentation for real-time transaction processing. We take this approach, since most existing commercial database systems do not support real-time scheduling or concurrency control. Neither do popular operating systems, e.g., Linux, underlying database systems. Ideally, however, one can replace FCFS and 2PL with real-time transaction scheduling and concurrency control mechanisms and provide them as a library to address both performance and deployment issues. A through investigation is reserved for future work.

2.1 Feedback Control Procedure

Our feedback controller shown in Figure 2 aims to support the desired response time bound D_s. Suppose that the database system started to operate at time 0 and let SP represent the sampling period, e.g., 1s, for feedback control. The k^{th} ($k \geq 1$) sampling *period* is the time interval $[(k-1)SP,\ kSP]$ and the k^{th} sampling *instant* is equal to kSP in the time domain. The performance monitor computes the error $e(k) = D_s - d(k)$ where $d(k)$ is the average response time of the transactions and queries finished in $[(k-1)SP,\ kSP]$. At the k^{th} sampling instant, the controller computes $\delta q(k)$, which is the queue size adjustment needed to support D_s in $[kSP,\ (k+1)SP]$, based on $e(k)$. A description of the overall feedback control procedure follows.

Step 1. At the k^{th} sampling instant, the delay controller computes the service delay error $e(k) = D_s - d(k)$.

Step 2. After computing $e(k)$, the controller computes the control signal $\delta q(k)$ based on $e(k)$. $\delta q(k)$ will be applied to existing transactions and transactions arriving at the RTDB in the time period $[kSP,\ (k+1)SP]$. Under overload, $\delta q(k) < 0$ and Chronos needs to reduce the backlog due to data/resource contention. If $\delta q(k) \geq 0$, jump to Step 4. A detailed discussion of the controller design is given in Section 3.

Step 3. If $\delta q(k) < 0$, apply the adaptive update policy to a subset of cold data whose update frequencies are higher than their access frequencies. As a result, the period $p[i]$ of a cold data item i can be increased to the new period $p[i]_{new}$. Accordingly, increase $\delta q(k)$ by $(p[i]_{new} - p[i])/p[i]$, since the update period extension has a similar effect to reducing the service request rate. Repeat this step until $\delta q(k) \geq 0$ or no more freshness adaptation is possible. A detailed discussion of the adaptive update policy is given in Section 2.2.

Step 4. The new ready queue size to support D_s in the time interval $[kSP,\ (k+1)SP]$ is: $q(k) = q(k-1) + \delta q(k)$. Thus, the ready queue size will decrease when $\delta q(k) < 0$ and vice versa. We apply the anti-windup technique [23] to bound the queue size between 0 and the max_qsize defined by the database system; that is, if $q(k) < 0$ we set $q(k) = 0$. Also, we make $q(k) = max\_qsize$ if $q(k) > max\_qsize$. Chronos applies admission control to transactions arriving in $[kSP,\ (k+1)SP]$. Hence, incoming transactions will be dropped upon their arrivals, if the queue length is already equal to or larger than $q(k)$ due to the backlog.

2.2 Freshness Adaptation

Under overload, update workloads can be reduced based on the notion of *flexible validity intervals* [8] to gracefully relax absolute validity intervals [18] designed to maintain the temporal consistency of data. If data d_i's absolute validity interval is $avi[i]$, its update period $p[i] = 0.5avi[i]$ to support the freshness of d_i [18]. For cost-effective temporal data updates, the access update ratio $AUR[i]$ is computed for each temporal data d_i based on the update frequency, i.e., $1/p[i]$, and access frequency [8]:

$$AUR[i] = \frac{Access\ Frequency[i]}{Update\ Frequency[i]} \quad (1)$$

d_i is *hot* if $AUR[i] \geq 1$; otherwise, it is *cold*.

In this paper, we develop a novel self-adaptive freshness management policy activated by temporal data update and access patterns as well as the system status. Our adaptive update policy can converge to the balance where $AUR[i] = 1$ for an arbitrary temporal data d_i in the database; that is, d_i is updated just as often as it is accessed. By achieving the balance, temporal data updates are optimized by naturally differentiating the freshness of temporal data according to user transaction needs. If a data is frequently accessed, it will be updated frequently and vice versa. Also, this approach is not specific to a single application but applicable to general real-time data services. A detailed description of the adaptive update policy follows.

Under overload, we increase the update period of a cold data d_i:

$$p[i]_{new} = min\Big(\frac{p[i]}{AUR[i]}, P_{max}\Big) \quad (2)$$

where P_{max} is the update period relaxation bound that can be determined by an administrator of an application such as stock trading. As $AUR[i] < 1$ for cold data d_i, its update period will increase to $p[i]/AUR[i]$ as long as $p[i]/AUR[i] \leq P_{max}$ according to Eq 2. We consider P_{max}, since an application administrator may want the update period of cold data d_i to be below P_{max} to avoid that the update period of d_i increases with no bound when it is rarely accessed. This approach can quickly react to time-varying freshness requirements, for example, when previously cold stock items become popular due to market status changes. In Eq 2, we take the minimum between $p[i]/AUR[i]$ and P_{max} to ensure that $p[i]_{new} \leq P_{max}$. In this paper, we set $P_{max} = 5s$ to ensure that every stock price is updated at least every 5s. After the update period of d_i increases, Chronos adjusts the fvi (flexible validity interval) for d_i: $fvi[i]_{new} = 2p[i]_{new}$ where $fvi[i] = avi[i]$ initially. Further, $avi[i] \leq fvi[i]_{new} \leq 2P_{max}$. Similarly, when the system is underutilized the freshness can be upgraded for hot data. In this paper, we only evaluate the freshness degradation, since we mainly consider overload conditions. In the future, we will analyze the performance impact of freshness upgrades too.

By increasing the update period under overload, one can reduce potential data/resource contention between temporal data updates and user transactions too, expediting overload management. This phenomenon is observed in our performance evaluation in Section 4. To degrade the freshness under overload, Chronos needs to linearly search the first cold data item in the database and increase its update period according to Eq 2. Thus, the worst case time complexity is $O(N)$ where N is the number of the temporal data in the database.

Our approach contrasts to the previous work [8] in which data are sorted in descending order of AUR and the update period of the cold data with the lowest AUR is increased first by a *fixed, systemwide* amount. This is repeated to the next lowest AUR data and so on. Hence, the time complexity of the approach proposed in [8] is $O(NlogN)$ for sorting AUR values. Further, freshness degradation is not differentiated considering update and access patterns.

In this paper, we consider an example RTDB QoS specification **QoS-SPEC** $= \{D_s = 2s, D_o = 2.5s, D_t = 100s, P_{max} = 5s\}$ that can be specified by an administrator of a real-time data service application such as e-commerce. According to the QoS-SPEC, the service delay needs to be smaller than or equal to $D_s = 2s$. An overshoot, if any, is the transient response time longer than D_s. It is desired that D_o is smaller than or equal to $2.5s$. Supporting $D_o \leq 2.5s$ is challenging in a database, potentially involving severe data and resource contention. The settling time D_t is the time taken for Chronos to handle an overshoot, if any, and decrease the delay back to D_s. In this paper, the settling time D_t is desired to be shorter than or equal to $100s$. In general, an overshoot and settling time have trade-off relations [15]. In this paper, we aim to support a small overshoot. If an overshoot is small, a relatively long settling time can be acceptable. In Section 4, we show Chronos can closely support this stringent QoS specification given bursty workloads. In the future, we will also consider other QoS specifications to further evaluate the performance of our database QoS management scheme consisted of feedback control, adaptive updates, and admission control.

3 RTDB Modeling and System Identification

In this section, we take a systematic approach to designing and tuning the data service delay controller. Due to the systematic modeling and tuning described in this section, we can meet QoS-SPEC via feedback control. Section 3.1 discusses our open-loop database model via the relation between the ready queue size and service delay. In Section 3.2, system identification (SYSID) techniques [15, 6] are applied to derive the model parameters. Using the SYSID results, the closed-loop transfer function is derived in Section 3.3. Using the closed-loop transfer function, the controller is tuned via the Root Locus method [15] to support the desired average and transient service delay specified in QoS-SPEC, while supporting the stability of the closed-loop system.

3.1 Queue Length vs. Response Time

In this paper, we derive a RTDB model in the discrete time domain using the ARX (Auto Regressive eXternal) model [15, 6], which is widely used to identify computational system dynamics [6, 10, 11]. Specifically, we model the relationship between the data service delay $d(k)$ and the delay $d(k-i)$ and queue size $q(k-i)$ measured in the k^{th} and previous sampling periods:

$$d(k) = \sum_{i=1}^{n} \{a_i d(k-i) + b_i q(k-i)\} \tag{3}$$

where $n(\geq 1)$ is the system order determining the size of the historical system dynamics data used for SYSID. The unknown model parameters a_i's and b_i's in Eq 3 can be derived via system identification [15, 6].

3.2 System Identification

The objective of our SYSID is to minimize the sum of the squared errors for the data service delay prediction based on the ready queue size. For SYSID, 2,400 client threads concurrently send data service requests to Chronos for one hour. Each client sends a service request and waits for the response from Chronos. After receiving the transaction or query processing result, it waits for an inter-request time. The inter-request time is randomly selected in a range [1s, 3s]. Note that this is a large operating range: If every client has a 3s inter-request time, the clients submit 800 tps (transactions per second) in total. They submit 2,400 tps if all of them have a 1s inter-request time. In this way, we can model high performance real-time data services dealing with widely varying workloads. Our control modeling and tuning are valid in this operating range. Beyond the operating range, a feedback controller may or may not support the desired performance [15, 6]. Analyzing the robustness of our feedback controller for workloads outside the operating range is reserved for future work.

For SYSID, no feedback control, admission control, or freshness adaptation is applied. Thus, Chronos accepts all incoming data service requests and updates each temporal data at the highest frequency. Specifically, each stock price is updated at every 0.5s for SYSID, which is equal to 6,000 stock price updates per second in total. The least square estimator predicts the response time at every 10s based on the queue length vs. response time statistics data observed in the SYSID window whose size is determined by the system order defined in Eq 3. In 10s, approximately $8,000-24,000$ transactions arrive at the database. By using a large number of data for SYSID, we aim to moderate the stochastic nature of a database as recommended in [6].

We have performed SYSID for the first order to fourth order system models. To analyze the accuracy of the SYSID results, we use the root mean square error (RMSE), which is a common accuracy metric for SYSID [15, 6]. Unfortunately, in our one hour SYSID, the $RMSE = \sqrt{\sum_{i=1}^{360} e^2(i)/360} > 5s$ for the first order to fourth order system models. The large RMSEs are unacceptable, since the desired delay bound is 2s in QoS-SPEC. The large prediction errors are mainly due to the wide operating range expressed by the range of the inter-request time as described before.

Since a large number of clients with varying inter-request times are prevalent in transaction processing, we consider an alternative approach for more accurate SYSID. Specifically, we use the square root values of the performance data, similar to [5]. The corresponding $RMSE = 0.66s$ for the *second order* model, which is nearly an order of magnitude better than the RMSE values of the previous SYSID results using the raw queue size and response time values. Another accuracy metric $R^2 = 1 - \frac{variance(response\ time\ prediction\ error)}{variance(actual\ response\ time)} = 0.79$. Usually, a model with $R^2 \geq 0.8$ is considered acceptable [6]. Considering the $RMSE$ and R^2, the SYSID results using the square root values of the ready queue size and response time can be considered acceptable.

We rejected the first order model due to its poor $RMSE$ and R^2 values. The third and fourth order models can only improve the $RMSE$ and R^2 by approximately 0.02 compared to the second order model. Thus, we choose the second order model due to the relatively low complexity. The derived second order model is:

$$\begin{aligned} d(k) = 0.507d(k-1) - 0.089d(k-2) + \\ 0.115q(k-1) + 0.009q(k-2) \end{aligned} \tag{4}$$

where $d(k)$ is the delay predicted to be observed in the k^{th} SYSID period based on the actual delay and queue length values measured in the $(k-1)^{th}$ and $(k-2)^{th}$ SYSID periods.

3.3 Data Service Delay Controller

To design the response time controller, we derive the transfer function of the open-loop Chronos database by taking the z-transform [15, 6] of Eq 4 to represent the relationship between the ready queue length and service delay in an algebraic manner:

$$D(z) = 0.507z^{-1}D(z) - 0.089z^{-2}D(z) + 0.115z^{-1}Q(z) + 0.009z^{-2}Q(z) \qquad (5)$$

where $D(z)$ is the z-transform of $d(k)$ and $Q(z)$ is the z-transform of $q(k)$ in Eq 4. After some algebraic manipulation, we get the following transfer function:

$$G(z) = \frac{D(z)}{Q(z)} = \frac{0.115z + 0.009}{z^2 - 0.507z + 0.089} \qquad (6)$$

Given the transfer function in Eq 6, we design the data service delay controller using a PI (proportional and integral) controller. We select a PI controller, since a P controller by itself cannot remove the steady state error [15, 6]. An I controller can eliminate the non-zero steady-state error when combined with a P controller [15, 6]. We do not use a derivative controller, because it is sensitive to noise [15, 6] and real-time data service workloads potentially involve bursty transaction arrivals and data conflicts. An efficient PI control law used by our response time controller to compute the control signal $q(k)$ at the k^{th} sampling instant is:

$$q(k) = q(k-1) + K_P[(K_I + 1)e(k) - e(k-1)] \qquad (7)$$

where the error $e(k) = D_s - d(k)$ at the k^{th} sampling period as shown in Figure 2. Given that, $\delta q(k) = q(k) - q(k-1)$ is the queue length adjustment needed to support the desired response time D_s in $[kSP, (k+1)SP]$. The transfer function of a PI controller can be obtained by taking the z-transform of Eq 7 to describe the relation between the controller input $e(k)$ and the controller output $\delta q(k)$:

$$C(z) = \frac{\alpha(z-\beta)}{z-1} \qquad (8)$$

where $\alpha = K_P(K_I + 1)$ and $\beta = 1/(K_I + 1)$. We need to carefully choose K_P and K_I to support the desired average and transient performance specified in QoS-SPEC.

To tune K_P and K_I, we need to derive the closed-loop transfer function. One can derive the closed loop transfer function using the transfer functions of the controller and the controlled system such as the database [15, 6]. Given the Chronos transfer function $G(z)$ (Eq 6) and PI controller transfer function $C(z)$ (Eq 8), the closed loop transfer function $\mathcal{F}(z)$ can be obtained in a standard way [15, 6]:

$$\mathcal{F}(z) = \frac{C(z)G(z)}{1 + C(z)G(z)} \qquad (9)$$

In this paper, we set the sampling period $\mathcal{S}P$ = 1s, because several hundreds to thousands of transactions per second arrive at Chronos in our experiments. Thus, Chronos has to quickly react to support the desired response time bound. Given $\mathcal{F}(z)$ and $\mathcal{S}P$ = 1s, via the Root Locus method in MATLAB [15], we locate the closed-loop poles−the roots of the denominator in Eq 9−inside the unit circle to support the stability of the closed loop system, while supporting D_s, D_o, and D_t of QoS-SPEC. The selected closed loop poles are 0.496 and $0.282 \pm 0.354i$. The corresponding $K_P = 1.29$ and $K_I = 2.01$.

4 Performance Evaluation

In this section, we evaluate our approach and several baselines to see whether they can support QoS-SPEC described in Section 2. We describe experimental environments and settings followed by the average and transient performance results.

A Chronos server runs in a Dell laptop with the 1.66 GHz dual core CPU and 1 GB memory. Clients run in two Dell desktop PCs. One PC has the 3 GHz CPU and 2GB memory, while the other one has the same CPU and 4 GB memory. Every machine runs Linux with the 2.6.15 kernel. The clients and server are connected via a 100 Mbps Ethernet switch and they communicate using the TCP protocol. Each client machine runs 900 client threads. Therefore, 1800 client threads continuously send service requests to the Chronos server. For 60% of time, a client thread issues a query about stock prices, because a large fraction of requests can be stock quotes in stock trading. For the other 40% of time, a client uniformly requests a portfolio update, purchase, or sale transaction at a time. A transaction accesses between 50 and 100 data. Initially, each temporal data, i.e., stock price, is updated at every 0.5s.

One experiment runs for 15 minutes. To generate bursty workloads, we vary the load described in Section 3.2 as follows. At the beginning of an experiment, the inter-request time is uniformly distributed in [2.5s, 3s]. Thus, approximately $600 - 720$ transactions per second arrive at the Chronos server. At 5 minutes, the range of the inter-request time is suddenly reduced to [1.5s, 2s] to model bursty workload changes.

Table 1. Tested Approaches

Open	Pure Berkeley DB
AC	Ad-hoc Admission Control
FC-C	Feedback Control AC
FC-CU	Feedback Control AC + AUP

Hence, $900 - 1,200$ transactions per second arrive at the database server. As a result, the workload randomly increases by $25\% - 100\%$ at 5 minutes and it is maintained until the end of an experiment at 15 minutes. An alternative approach for generating bursty workloads is to increase the number of client threads, while using a fixed range of inter-request time. In this paper, we choose to change the inter-request time, because it has less overhead than creating a larger number of client threads on the fly. As a result, it can give more immediate impacts to the workload applied to the database server.

For performance comparisons, we consider the four approaches shown in Table 1. Open is the basic Berkeley DB [3] without any special performance control facility. Thus, it represents a state-of-the-art database system. Under overload, AC applies admission control to incoming transactions in proportion to the response time error. FC-C applies admission control to incoming transactions according to the control signal computed in our feedback loop. However, it does not apply adaptive temporal data updates. FC-CU applies both adaptive temporal data updates and admission control according to the control signal computed in our feedback loop. Except Open, the tested approaches close the connection with a client thread after sending a busy message to the client thread, if the TCP connection establishment itself takes longer than D_s. This approach is reasonable, because it is impossible to support D_s in this case. Five experimental runs are performed for each tested approach with different seeds for uniform random number generation. In this section, the average of 5 runs is presented with 90% confidence intervals.

4.1 Average Performance

The average response time for each approach is shown in Figure 3. Open's average response time is $4.02 \pm 0.53s$. The average response time of AC is $3.56 \pm 0.83s$. FC-C and FC-CU achieve $3.22 \pm 0.87s$ and $1.26 \pm 0.42s$, respectively. From these results, we observe that only FC-CU can support the desired response time bound $D_s = 2$s specified in QoS-SPEC. By accepting all incoming transactions (and queries), Open becomes overloaded, failing to support D_s. AC

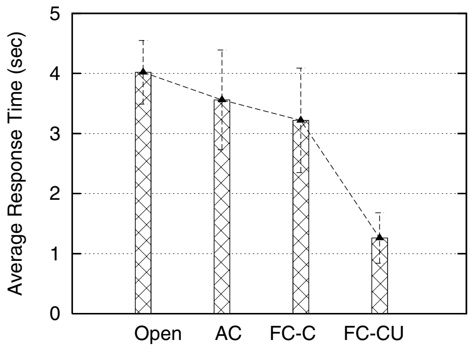

Figure 3. Average Response Time

shows the better response time than Open. Via systematic feedback control, FC-C provides a shorter response time than AC and Open; however, it cannot support D_s. From this, we observe that admission control by itself is insufficient for managing database overloads, involving not only physical resource but also data contention. In FC-CU, adaptive temporal data updates can reduce data/resource contention between user transactions and stock price updates, resulting in more effective overload management as discussed before. The cost of the adaptive update policy in FC-CU is the freshness reduction. In average, $60\% - 84\%$ of the update periods are extended to $1s - 4.3s$ in our experiments, while the remaining stock price data are updated at every 0.5s. Thus, $P_{max} = 5s$ in QoS-SPEC is supported.

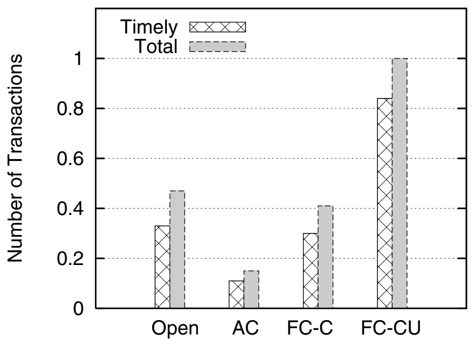

Figure 4. Normalized Number of the Committed Transactions and Queries

As shown in Figure 4, FC-CU processes the largest number of transactions among the tested approaches. It has processed more than 434,000 transactions in 15 minutes. Out of the committed transactions, more than 366,000 transactions are processed within D_s. Thus, in FC-CU, approximately 84% of the committed transactions finish within the desired delay bound. Figure 4 shows the total number of the committed transactions and the number of timely transactions finishing within D_s normalized to the corresponding numbers for FC-CU. In the figure, FC-CU significantly outperforms Open, AC, and FC-C. AC processes fewer transactions in time than Open does. FC-C's performance is close to Open, but much worse than FC-CU's. From these results, we observe that admission control is insufficient for overload management in Chronos as discussed before. Ad hoc admission control is even less effective. We have observed that it admits either too many or two few transactions, showing oscillating performance.

4.2 Transient Performance

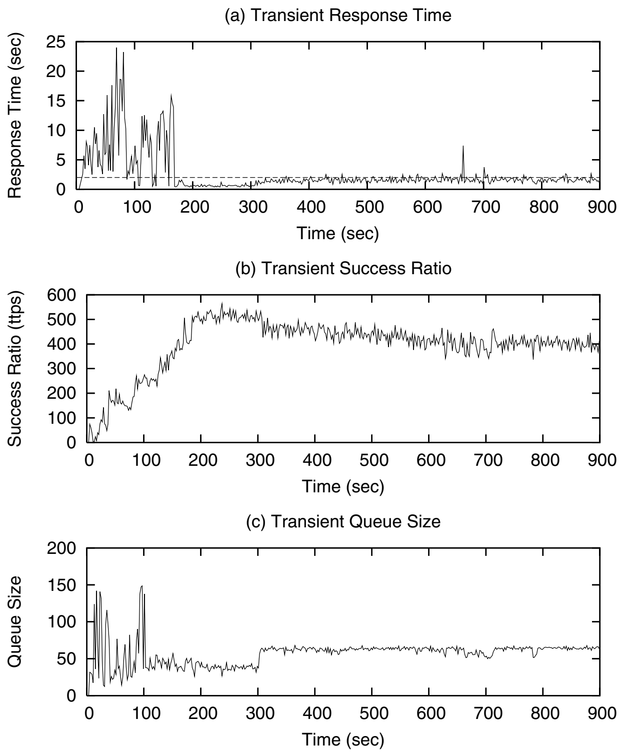

Figure 5. Transient Performance of FC-CU

In this subsection, we present the transient performance of FC-CU and the tested baselines. Figure 5(a) and Figure 6 show the transient response time of the tested approaches, which is measured at every 1s sampling period. FC-CU substantially outperforms the baselines in terms of the transient response time as shown in the figures. The response time overshoot of Open is 55.32s and its transient response time is over 15s for more than half of the experiment as shown in Figure 6(a). AC and FC-C in Figures 6(b) and 6(c)

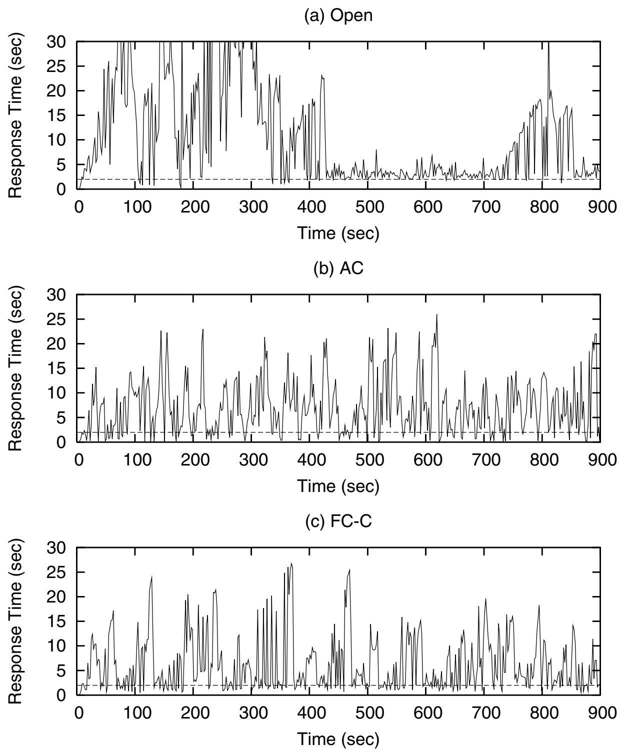

Figure 6. Transient Response Time of the Baselines

considerably reduce the transient response time compared to Open, but they frequently violate the desired delay bound by large amounts.

At the beginning in Figure 5(a), the response time is high mainly due to database initialization overhead involving a lot of disk accesses and data structure initialization. From 170s, however, its transient response time is below 2s, i.e., the horizontal bar in Figure 5(a), for most of the time. There are two relatively large overshoots in the time period between $600s - 700s$, but these overshoots decay in less than three sampling periods, satisfying $D_t \leq 100s$ in QoS-SPEC.

Figure 5(b) shows FC-CU's transient success ratio measured by the number of timely transactions per second (ttps), which denotes the number of transactions finishing within D_s at every second. Also, the transient ready queue length of FC-CU is plotted in Figure 5(c). In Figure 5(b), the success ratio is relatively low during the database initialization. As a result, the ready queue builds up, increasing the transient response time at the beginning as shown in Figures 5(b) and 5(c). Note that this may not be a serious problem, since a real stock trading database may choose to not accept service requests during the initialization phase. Further, the initialization takes less than 3 minutes in Chronos as shown in Figure 5(a). After the initialization, FC-CU achieves the generally increasing success ratio until 200s, while processing the backlog accumulated during the initialization as shown in Figure 5(b). In the figure, its success ratio ranges between 370 ttps $-$ 563 ttps in [170s, 900s]. The tested baselines achieve much lower success ratios compared to FC-CU due to widely oscillating transient response time shown in Figure 6. The maximum transient success ratio of Open is 332 ttps. AC's maximum success ratio is only 202 ttps, while FC-C's maximum success ratio is 290 ttps. Note that FC-CU's maximum success ratio is over 560 ttps as described before.

As shown in Figure 5(c), FC-CU maintains the queue size in a steady manner except the initialization period. The maximum queue size from the beginning to 108s is 149 data service requests as shown in the figure. It ranges between $26 - 58$ requests during the time period of [109s, 300s]. Note that the queue is not empty, since the feedback control signal is positive to accept more requests during this time interval where the response time is shorter than the desired 2s bound. After that, the queue length ranges between $49 - 71$ requests. The queue size has been increased at 300s, since the inter-request time is considerably reduced at 300s in our experiments. From this, we can observe that our feedback controller can effectively manage the queue size to support the desired delay bound. In summary, FC-CU significantly enhances the average and transient data service performance compared to the baselines by applying feedback control and efficient workload adaptation techniques.

5 Related Work

Most RTDB work is based on simulations [19]. Very little prior work on real-time data management [1, 9, 14, 7] has actually been implemented and evaluated in real database systems. However, no RTDB testbed is publicly available. As a result, it is hard to perform RTDB research in a real database system. To address this problem, we have developed Chronos [7], which is significantly extended in this paper as discussed before.

Conceptually, our work is closest to the recent work on feedback control in RTDBs, including [2, 8, 13], aiming to support the specified CPU utilization, deadline miss ratio, or data and timeliness imprecision in RTDBs. In this paper, we develop a new control model to support the desired response time for real-time data service requests. Our work is also different from the existing work in that our work is implemented and evaluated in a real database system.

In this paper, we propose a new temporal data update policy, which can efficiently manage the freshness in a self-adaptive manner different from our previous work [8]. Adelberg et. al. [1] observe that there is a trade-off between transaction timeliness and data freshness. The data imprecision and similarity concept [2, 4] can be applied to reduce update workloads under overload. When a certain data value is changed by less than the specified threshold, the corresponding update can be dropped under overload. Those approaches are complementary to our approach. RTDB performance could be further improved, if our adaptive update policy is augmented by the data similarity or imprecision concept. This is reserved for future work.

UNIT [17] and QUTS [16] propose adaptive query and update scheduling algorithms to optimize the user preference and profit, respectively. However, they do not aim to support the delay bound critical for real-time data services. Schroeder et. al. [20] determine, via queuing and control theoretic techniques, an appropriate number of transactions allowed to concurrently execute in a database system. However, their work aims to support neither the desired response time bound nor the data freshness essential for real-time data services. It lacks a formal discussion of controller design and stability analysis too.

Feedback control has been applied to manage the performance of various systems such as a web server and real-time middleware [10, 12, 11]. These approaches, however, may not be directly applicable to RTDBs, because they do not consider RTDB-specific issues such as the data freshness.

6 Conclusions and Future Work

As database workloads may considerably vary due to dynamic data/resource contention, it is challenging for a RTDB to process transactions in a timely manner using fresh data, e.g., current stock prices or traffic data. Also, transaction timeliness and data freshness requirements may compete for system resources. In this paper, we propose a novel feedback control model to support the desired data service delay by managing the amount of the backlog in the database. We also propose a new adaptive update policy to adapt the freshness of cold data in an efficient way based on temporal data access and update patterns. Unlike most existing work on feedback control of RTDB performance, we actually implement and evaluate feedback control and database workload adaptation techniques in a real database testbed. In our stock trading testbed experiments, our feedback control system supports the desired response time bound and data freshness, while processing a significantly larger number of transactions in time compared to the tested baselines. Thus, in this paper, we show the feasibility of feedback control of a real database system. In the future, we will continue to enhance feedback control, data service quality adaptation schemes, transaction scheduling, and concurrency control schemes.

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, J. Hansson, and S. H. Son. Specification and Management of QoS in Real-Time Databases Supporting Imprecise Computations. *IEEE Transactions on Computers*, 55(3):304–319, 2006.

[3] Oracle Berkeley DB Product Family, High Performance, Embeddable Database Engines. Available at http://www.oracle.com/database/berkeley-db/index.html.

[4] D. Chen and A. Mok. SRDE-Application of Data Similarity to Process Control. In *the 20th IEEE Real-Time Systems Symposium*, 1999.

[5] N. R. Draper and H. Smith. *Applied Regression Analysis*. Wiley, 1968.

[6] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. A John Wiley and Sons, Inc., Publication, 2004.

[7] K. D. Kang, P. H. Sin, J. Oh, and S. H. Son. A Real-Time Database Testbed and Performance Evaluation. In *the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[8] 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.

[9] 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.

[10] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium*, page 51, 2001.

[11] C. Lu, X. Wang, and C. Gill. Feedback Control Real-Time Scheduling in ORB Middleware. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 37, 2003.

[12] Y. Lu, T. F. Abdelzaher, C. Lu, L. Sha, and X. Liu. Feedback Control with Queueing-Theoretic Prediction for Relative Delay Guarantees in Web Servers. In *the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[13] J. Oh and K. D. Kang. An Approach for Real-Time Database Modeling and Performance Management. In *the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[14] C.-S. Peng, K.-J. Lin, and C. Boettcher. Real-Time Database Benchmark Design for Avionics Systems. In *the First International Workshop on Real-Time Databases: Issues and Applications*, 1996.

[15] C. L. Phillips and H. T. Nagle. *Digital Control System Analysis and Design (3rd edition)*. Prentice Hall, 1995.

[16] H. Qu and A. Labrinidis. Preference-Aware Query and Update Scheduling in Web-databases. In *the 23rd International Conference on Data Engineering*, 2007.

[17] H. Qu, A. Labrinidis, and D. Mosse. UNIT: User-centric Transaction Management in Web-Database Systems. In *the 22nd International Conference on Data Engineering*, 2006.

[18] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[19] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-Time Databases and Data Services. In *Real-Time Systems*, volume 28, Nov.-Dec. 2004.

[20] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proceedings of the 22nd International Conference on Data Engineering*, page 60, 2006.

[21] Transaction processing performance council. http://www.tpc.org/.

[22] U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of E-Commerce Traffic. *Electronic Commerce Research*, 3(1-2), 2003.

[23] D. Vrancic. *Design of Anti-Windup and Bumpless Transfer Protection*. PhD thesis, University of Ljubljana, Slovenia, 1997.

[24] 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.

[25] Yahoo! Finance. http://finance.yahoo.com/.