Object-Oriented Design of Main-Memory DBMS for Real-Time Applications

School of Electrical Eng. and ERC for Advanced Control and Instrumentation
Seoul National University
San 56-1, Shillindong, Kwanakku, Seoul, 151-742, Korea
E-mail: [chask, parkbd, sjlee, shsong, jhpark@kdb.snu.ac.kr]

J. S. Lee, S. Y. Park, D. Y. Hur
Electronic and Telecommunication Research Institute
P.O.Box 106, Yusong, Taejon, 305-600, Korea

G. B. Kim
Pohang Iron and Steel Company
P.O.Box 36, Pohang, 790-785, Korea

Abstract

Many applications, such as telecommunication, process control, and virtual reality, require real-time access to database. Main-memory DBMS, which becomes feasible with the increasing availability of large and relatively cheap memory, can provide better performance than disk-based systems for real-time applications. This paper presents an overall architecture of M²RT, a Main-Memory Real-Time DBMS, and an object-oriented design of its storage system called M²RTSS. M²RTSS provides classes that implement the core functionality of storage management, real-time transaction scheduling, and recovery. Implementation-specific information is encapsulated in these classes and extensions can be made by inheritance. With object-oriented features, M²RTSS can easily incorporate new development in application requirements and the result of ongoing research in real-time systems.

Keywords: object-oriented design and implementation, extensibility, main-memory DBMS, real-time DBMS

1 Introduction

Many applications, particularly, in telecommunication, process control, and virtual reality, require real-time access to database[1]. Such applications typically require high transaction rates, coupled with bounded latency for transactions. Disk-based database systems, however, are inadequate for this type of applications because the performance is physically limited by the mechanical movement of disk arms. The increasing availability of large and relatively cheap memory makes it feasible to store database entirely or almost entirely in memory. With data directly accessed from memory, main-memory database can provide better performance for real-time applications.

There have been many designs and implementations of main-memory DBMS[2]. These include IMS/VS Fast Path[3], TPK[4], OBE[5], System M[6], and main-memory storage systems, such as the Starburst’s memory-resident storage component[7] and Dali[8]. These systems, however, were designed to maximize transaction throughput rather than to meet the timing constraints of individual transactions.

This paper describes an overall architecture of a Main-Memory Real-Time DBMS, called M²RT, and an object-oriented design of M²RTSS, the storage system of M²RT. M²RT is being developed based on our previous experience on DBMS implementation. This includes SNU’s design of main-memory object-oriented DBMS and ETRI’s shared-memory implementation of a storage system called FLASH[9]. Our current focus is on M²RTSS. The primary design goal of M²RTSS is its extensibility to incorporate easily advances in related hardware and software technology and new development in application requirements. For example, stable memory devices such as NVSIMM[10] are becoming easily affordable, and we expect M²RTSS to take advantage of such development in hardware technology to achieve high perfor-
mance logging. To provide this extensibility, M\textsuperscript{2}RTSS takes an object-oriented approach\cite{11,12} in its design and implementation. The core functionality of M\textsuperscript{2}RTSS is encapsulated in various classes provided by the system, and extensions can be made by inheritance.

This paper is organized as follows. Section 2 presents an overview of M\textsuperscript{2}RT architecture and the thread architecture of M\textsuperscript{2}RTSS. Following sections discuss M\textsuperscript{2}RTSS in detail. Section 6 summarizes this paper.

2 Architecture

2.1 M\textsuperscript{2}RT Layers and Components

As shown in figure 1, M\textsuperscript{2}RT is roughly divided into two layers: M\textsuperscript{2}RTSS and end-user's interactive and programming interface. M\textsuperscript{2}RTSS provides real-time transaction scheduling as well as core functionality of main-memory DBMS such as memory management, concurrency control, and recovery. This functionality is supported by several managers, each of which is implemented as an object. Following sections describe the managers in detail.

On top of M\textsuperscript{2}RTSS, RT-SQL engine implements the relational query processing module. RT-SQL is an extended version of standard SQL, in which users can specify the timing constraints for queries. Interactive RT-SQL provides an interactive query interface for end-users. For programmers, Embedded RT-SQL/C++ is provided. Programmers can also access M\textsuperscript{2}RTSS directly through C++ API. In the future, this API is expected to be used as the basis of supporting object-oriented data model, such as an extended version of ODMG\cite{13} for real-time applications.

2.2 M\textsuperscript{2}RTSS

M\textsuperscript{2}RTSS consists of several managers. Figure 2 shows these managers and the interaction among them. The transaction manager is responsible for the interaction with the upper layer, and coordinates other managers to support transaction begin, commit, and abort. The transaction scheduler determines the execution order of transactions. The storage manager manages the primary database in main memory and provides routines for creating, deleting, and accessing records, indices, etc. The log manager and the checkpoint manager make main-memory database recoverable. The lock manager implements concurrency control. M\textsuperscript{2}RTSS is currently implemented in C++ on Solaris 2.4 running on SUN SparcServer 1000.

The server process is multi-threaded to serve several requests simultaneously and to perform disk I/O in parallel with normal transaction processing. When M\textsuperscript{2}RTSS starts up, it creates a pool of threads using Solaris 2.4 threads package. A thread is not tied to any particular user process or transaction. Any request from user processes is transformed to an action object, the unit of work that is executed by each thread. From the perspective of the server process, a user transaction is a sequence of action objects. Figure 3 shows this thread architecture.

When a request arrives in the input queue, the action dispatcher thread converts it into an action object and assigns it to an available action processing thread through the action queue. An action processing thread runs until it has to wait for a resource such as lock
or semaphore, or is preempted by thread scheduling. When it finishes executing an action object, it puts the result on the output queue and awaits another action object.

The checkpoint thread and the log flush thread deal with I/O and run asynchronously with the other threads. The former migrates the dirty pages in the primary database to backup, and the latter flushes the log tail in log buffer to stable log volume.

3 Database Structure

The primary database of M²RTSS consists of a large number of objects, such as containers, indices, catalogs, and a schema. Each type of objects encapsulates its own data and provides functions that manipulate the data to other objects.

The storage manager coordinates the interaction among the objects of the primary database, and provides an unified interface to other managers and users. The functions of the storage manager include creation/deletion of a container/index, insertion/deletion/update of an entry of the container/index, and searching a record with value by using indices or by scanning the container.

3.1 Physical Structure

The memory space of the primary database has hierarchical structure. In detail, the primary database is divided into a certain number of fixed-size segments, each of which is partitioned into a number of pages. All pages are split into a number of slots.

Segment is the unit of memory allocation, hence the primary database can grow stepwise in the size of a segment. Segment is also the unit of checkpointing. Each segment is classified into system segment, container segment, index segment, and temporary segment according to the type of objects stored and the necessity of checkpointing. System segments contain the objects that are maintained by M²RTSS, such as catalogs and schema, and their changes are reflected on the backup database. Container segments, and index segments store the user-created containers, indices respectively, and their changes are also written on the backup database. Temporary segments manage the containers and indices that are created by system as the intermediate results. Their changes, however, are not reflected on the backup database.

The pages serve as the building block of the objects of the primary database, and are classified into several types according to their structure and size. The pages that are partitioned into fixed-size slots are used for catalogs and indices which entries are fixed-length. The pages that are divided into variable-size slots are employed for containers and schema which entries are variable-length. The size of a page can be 4, 8, 16, 32, and 64 KB in current implementation.

3.2 Logical Components

In figure 4, the objects of the primary database and their relationships are shown. A container is a collection of the records with a fixed number of fields and with fixed data types. Hence, a single record descriptor is enough to describe all records in a container. Each record in a container, however, can be variable in length because data types of variable-length, such as zero-terminated-string are supported. Pointers to records are maintained through an identifying scheme, named RID(Record IDentifier) in which an identifier

![Figure 3: Thread Architecture](image3)

![Figure 4: Logical Structure of a Database](image4)
is composed of the segment number, the page number, and the slot number. A RID of a slot is unique, is converted into the virtual address of the slot by simple calculation.

M$^2$RTSS supports hash indices for exact value searches and trees for range queries. The hash index, named ECBH (Extended Chained Bucket Hashing) [9], is a combination of extended hashing and chained bucket hashing, and is expected to give good performance for exact search for unordered records. The tree index, named T-tree [14], provides excellent overall performance for range search at low cost in storage space. Hashing and tree are designed by using inheritance, hence, other hashing or trees, are easily adapted to M$^2$RTSS. Multiple indices may be built over the same container and an index may be specified over multiple fields. In a traditional disk-based system, an index entry consists of a key value and the identifier of the record containing the key. To save space, M$^2$RTSS index entry consists of only RID to the record because the key value can be easily extracted from the memory-resident record.

M$^2$RTSS maintains several types of objects such as a schema, and catalogs. The schema is the collection of the record descriptors of all containers of the primary database. The catalogs are the collection of the control information of containers, ECBH index, and T-tree index respectively. The container catalog maintains the name and the position of all containers, and the RIDs to the entries of index catalog. An entry of the ECBH catalog and the T-tree catalog contains the control information of an index, such as the name of index, the key fields description, and the RID of the first index entry.

## 4 Real-Time Transaction Scheduling

In M$^2$RTSS, transactions are executed by Solaris 2.4 threads with their priorities assigned by the real-time scheduler. The real-time transaction scheduling of M$^2$RTSS is preemptive, since the Solaris 2.4 kernel provides preemptive priority-based scheduling of threads [15].

A real-time transaction scheduling algorithm should maximize both concurrency and resource utilization subject to three constraints: data consistency, transaction correctness, and transaction deadlines [16]. Satisfying timing constraints while preserving data consistency requires the integration of concurrency control protocols and real-time scheduling protocols. In this section, we present M$^2$RTSS’s two forms of scheduling protocols: management of priorities and conflict resolution, which are essential for predictability and responsiveness of real-time database systems.

### 4.1 Priority Assignment

As a real-time application has different characteristics and requirements from system to system, priority assignment protocol should be implemented according to what kind of application will be used on top of the system. For example, in a system to initiate trades in a stock market, the timing constraints of a transaction is combined with its criticalness to take the form of the priority of the transaction. In such a system, the criticalness of a transaction represents the benefit that might be obtained in case of being committed without violating its timing constraints. In other real-time systems which are used to respond to external stimuli (e.g. in combat systems) or to control physical devices (e.g. in auto pilot systems), reducing the deadline miss ratio is much more important than criticalness, since an out-of-date result is useless.

Most real-time systems are developed for some particular applications and their scheduling policies are fixed accordingly. M$^2$RTSS’s scheduler, however, is designed to handle various types of real-time transactions. The priority assignment scheme can be configured to meet the requirements of a specific real-time application.

#### 4.1.1 Aperiodic Transactions

In M$^2$RTSS, aperiodic real-time transactions are classified into two groups: soft and firm aperiodic real-time transactions. A firm real-time transaction is defined as a transaction which doesn’t have critical timing constraints but has no value when it does not meet these constraints [17]. Hence, the scheduler should abort a firm transaction when it misses its deadline.

The scheduler provides two alternatives of priority assignment policies: the earliest deadline and the least slack algorithm. A priority of a transaction should be mapped into an integer from 0 to maximum value, say n, since it is used for the thread scheduling of the Solaris 2.4 kernel, as stated previously. That is, a real priority is converted into an integer priority, say P. Programmers are provided with more alternatives with different characteristics according to converting scheme. For example, two kind of deadline-driven policies are provided. The one converts \((\text{deadline} - \text{current time})\) into a thread priority, which needs overall priority reassignment whenever a new real-time transaction enters. The other converts \((\text{deadline} - \text{arrival time})\) into a thread priority, which is faster but not optimal.

When the programmer wants to reflect the criticalness of a transaction to its priority, say P, the scheduler runs in the \textit{criticalness-priority} mode. In this mode,
\[
P_2 = w_1 \cdot P_1 + w_2 \cdot \text{criticalness},
\]
where \(0 \leq P_1\), \(\text{criticalness} \leq n\), \(0 \leq w_1, w_2 \leq 1, w_1 + w_2 = 1\)

4.1.2 Periodic Transactions

For periodic transactions, it is optimal to adopt a rate-monotonic priority scheme[18]. This algorithm assigns higher priorities to transactions with shorter periods. If periodic transactions should be jointly scheduled with aperiodic transactions, aperiodic transactions can be executed at lower priority level than periodic transactions, which are scheduled by rate-monotonic algorithm[19]. This scheme, however, may not be suitable if the application mainly generates aperiodic transactions, since a periodic transaction usually has much slack time. Hence, the programmer is provided with the other alternative: the scheduler doesn’t discriminate between periodic and aperiodic real-time transactions. That is, every transactions, whether periodic or not, will be scheduled according to the scheme described in the section 3.1.1.

The class definition of real-time transaction scheduler is as follows:

```java
class TxScheduler {
private:
  MODE_CR mode; // criticality-priority, normal
  float w1, w2;
  MODE_Policy alg1;
  MODE_Policy alg2;
  int setPriority(TX_TYPE t, time d, time a,
                   time e, int c, time T);
  int mapRealToInt(float p);
...
public:
  TxScheduler(MODE_PR a1, float a2, float a3,
              MODE_Policy a4, MODE_Policy a5);
  int schedule(Transaction *tx, TX_TYPE t, time d,
                 time e, int c, time T = 0);
...}
```

4.2 Concurrency Control

The concurrency control protocol provided by \(M^2\)RTSS is a form of two-phase locking(2PL)[20]. 2PL is a good choice because it detects and resolves conflicts early in the execution of a transaction. In an optimistic protocol[21], each transaction is executed to completion and then verified whether it conflicted with another transaction. If so, it is aborted and restarted. If conflicts occur frequently, it can be very wasteful to execute transactions only to abort them. Locking checks for conflicts every time a lock request is made. Conflicts can be resolved immediately before additional service is granted to transactions which are going to be restarted anyway.

Lock-based protocols can be classified into priority inheritance, high priority, and conditional restart protocols according to the algorithm of resolving priority inversion problem[20]. \(M^2\)RTSS currently uses the high priority protocol. With this strategy \(M^2\)RTSS resolves a conflict in favor of the transaction with the higher priority. \(M^2\)RTSS aborts the low-priority lock holder and lets the high-priority lock requester proceed. \(M^2\)RTSS also limits the number of transactions executing concurrently to reduce the probability of aborting due to data conflict.

The lock manager provides a simple but high level interface that hides underlying implementation details. Conflict resolving protocols, such as the high priority protocol, are encapsulated in it. As a result \(M^2\)RTSS can switch protocols and incorporate new algorithms easily. The methods tryLock and releaseLock of the lock manager are provided for concurrency control. The arguments to tryLock specify the container accessed, the access mode, and the lock timeout. The method releaseLock is implicitly executed on all items at the end of transaction.

It has been suggested that very large lock granule is acceptable for memory-resident data[22], hence \(M^2\)RTSS currently supports only container-level locking.

5 Recovery Management

Recovery-related managers deal with I/O and should be designed with care so that they do not impede the overall performance. To ensure the consistency and durability of the database in case of system or transaction failures, logging and checkpointing are performed during the normal operation. Logging notes on stable storage all updates done to the database, and checkpointing periodically creates a snapshot of the database on disk.

The conventional recovery schemes for main-memory database[22], [23], which are designed to maximize average transaction throughput, should be reconsidered from the perspective of real-time applications. For example, group commit[23],[24], which commits a group of transactions by each flush, is not suitable to minimize worst-case individual transaction times. The recovery subsystem is designed to explore the architecture and algorithms best suited for real-time applications. To provide an experimental testbed for comparing algorithms, \(M^2\)RTSS defines the com-
mon interface and makes it possible to switch among alternatives.

5.1 Log Manager

The log consists of two parts, a memory-resident buffer and a stable log volume that resides on disk. The
memory-resident buffer holds the log tail, the most recently created part of the log. Stable memory[22,24,25]
can be used as a write buffer between memory-resident buffer and disk. With stable memory, transactions can
commit without disk I/O.

The principal job of the log manager is to provide the interface to the log. The log manager appends log records
and undo/redo actions are performed by log records themselves. In other words, any specific algorithms or types of log
records are encapsulated and extensions can be made by inheritance. For example, logical logging which is necessary to support optimistic concurrency control can be supported as such.

The class declaration of log records is as follows:

class LogRec {
private:
  LSN   lsn;
  TxID  txid;
  Boolean flush; // Force or No Force
...
public:
  // any specific algorithms are
  // implemented at derived class
  virtual Undo();
  virtual Redo();
...}

5.2 Checkpoint Manager

The checkpoint manager is responsible for migrating changes in the primary database to the backup.
Checkpointing should interfere as little as possible with transaction processing. Transaction-consistent or
action-consistent checkpointing[26,27] require synchronization with transaction processing. On the other hand, fuzzy checkpointing[28] can be performed asynchronously.

In M2RTSS, as stated previously, the database is divided into segments. Different segments in the
database can have different checkpointing schemes as in [8]. For example, segments holding transient data
structures such as indices that are recomputable during recovery, does not need checkpointing at all. Currently, M2RTSS supports fuzzy checkpointing as its default.

6 Summary

This paper presented an overview of M2RT architecture, and described the current design of its storage system. The Object-oriented approach is taken in design and implementation to facilitate the extension of the system when it needs to be extended to meet a new set of application requirements or to take advantage of advances in related hardware and software technology.

In M2RTSS, the priority assignment scheme of the real-time transaction scheduler can be configured to
meet the requirement of a specific real-time application. The high priority protocol is assumed by the
lock manager to resolve the priority inversion problem. Specific algorithms or types of log records are encapsulated and extensions can be made by inheritance. By default, M2RTSS uses the fuzzy checkpointing algorithm.

M2RTSS is currently under implementation in C++ on Solaris 2.4. By using thread, M2RTSS is divided into several threads running concurrently, so that it can avoid problems, such as I/O blocking and inter-process communication overhead. We have a plan to run a performance test. Also planned is to employ it as an embedded DBMS for various real-time applications such as process control[29] and virtual reality.

REFERENCES


[10] A. Haq, “Non-Volatile Memory SIMM (NVSIMM),” the specification for Axil’s NVSIMM memory SIMM.


