MDARTS: A Multiprocessor Database Architecture for Real-Time Systems

Victor B. Lortz Kang G. Shin

Real-Time Computing Laboratory
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

{vbl,kgshin}@eecs.umich.edu
(313) 763-0391

ABSTRACT

Some of the advanced real-time systems being proposed, such as the Next Generation Workstation/Machine Controller (NGC) for automated factories, require a built-in database to support concurrent data access and provide well-defined interfaces between software modules. However, conventional database systems do not provide the performance levels or response time guarantees needed by real-time applications. To address this need, we are designing, implementing, and evaluating an object-oriented software system called Multiprocessor Database Architecture for Real-Time Systems (MDARTS). An important feature of MDARTS is that it supports explicit declarations of real-time requirements and semantic constraints within application code. The database examines these declarations at application initialization time and dynamically adjusts its data management strategy accordingly. The constraints serve as contracts between applications and the database system that help decouple application code from the database implementation. For maximum performance on shared-memory multiprocessors, MDARTS supports concurrent, direct, shared-memory data access. For data access with less stringent timing constraints, it also supports remote transactions across networks and provides interfaces to external database systems.

Keywords: real-time databases, object-oriented interfaces, exemplar-based programming, semantic constraints, concurrency control, shared memory, atomic data types

The work reported in this paper was supported in part by the Office of Naval Research under grant N00014-92-J-1080 and by the National Science Foundation Industry/University Cooperative Research Center at the University of Michigan.

1 Introduction

In recent years there has been a growing use of digital computers for real-time applications such as robot controllers, advanced manufacturing systems, air traffic control systems, nuclear reactor controllers, and "smart" weapons systems. The importance of these applications for economic competitiveness and public safety can hardly be overstated. In the manufacturing sector alone, the economic potential of automated factories is enormous. Before this potential can be realized, however, there are many technical challenges to overcome. One of the challenges is to equip real-time systems with databases to manage their data, often subject to stringent timing constraints. Therefore, it is critical that databases capable of supporting real-time applications be developed.

The chief difficulty in applying database technology to real-time systems is that conventional database architectures are not designed to provide the performance levels or response time guarantees needed by real-time systems. Most conventional database systems are disk-based and use transaction logging and two-phase locking protocols to ensure transaction atomicity and serializability. These characteristics preserve data integrity, but they also result in relatively slow and unpredictable response times. Thus, it is not feasible to simply connect a conventional multiuser database system, such as Oracle or Sybase, to a real-time manufacturing machine controller. The relatively slow and unpredictable database response times would cause time-critical tasks to miss deadlines [29, 27].

The inadequacy of conventional database systems for real-time applications has spawned the field of real-time databases. Real-time database systems are usually considered to be functionally equivalent to conventional database systems except that they provide fast, guaranteed response times [12, 24]. Although some progress in the field of real-time databases has been made, most of the work has focused on specific sub-problems such as transaction scheduling protocols. There are few successful real-time database implementations. Some database vendors claim to provide "real-time" capabilities, but by this they mean fast average-case performance and/or preemptive, priority-based transaction processing [11, 25]. These database systems are appropriate for soft real-time applications where it is desirable, but not crucial, to meet every deadline. However, providing fast average-case performance and transaction priorities is not sufficient for hard real-time applications such as high-speed control systems where each transaction *must* meet a strict deadline.

To our knowledge, no prior database implementation provides explicit deadline guarantees suitable for high-speed control systems. By deadline guarantees, we mean that individual database transactions are bounded in their execution times by application-specified limits or deadlines. For these guarantees to be useful in real-time systems, the execution time bounds must also be small enough for the application. A guaranteed one-second response time is useless when the task requires one millisecond response times.

To address the need for high-performance real-time databases, we are designing and implementing a new object-oriented real-time database architecture called Multiprocessor Database Architecture for Real-Time Systems (MDARTS). MDARTS supports explicit declarations of real-time requirements and semantic constraints within application code. MDARTS examines these declarations during application initialization and dynamically adjusts its data management strategy accordingly. Other researchers have recognized the potential for customizing real-time database services according to application semantics, but MDARTS is the first real-time database that provides a practical framework for communicating and utilizing such information. MDARTS is also extensible: new semantic constraints can be defined and integrated seamlessly with MDARTS without even requiring

recompilation of the MDARTS library.

For maximum performance on shared-memory multiprocessors, MDARTS supports direct, concurrent, shared-memory data access. The transaction processing overhead in MDARTS for direct shared-memory access is low – typically one or two ordinary procedure calls and a fetch operation over the common bus. On a 25MHz 68030-based shared-memory multiprocessor, our MDARTS implementation can guarantee transaction times of twenty microseconds to retrieve or store a 12-byte data object if locking is not needed. If application semantics require locking, performance guarantees depend upon the lock implementation and the number of tasks that can simultaneously request the lock. With three concurrent tasks running on separate CPUs performing updates to a 12-byte lockable data object, MDARTS can guarantee response times of 80 microseconds. These performance numbers are highly dependent upon the particular computing platform; MDARTS adjusts its performance guarantees according to the speed of the computing platform. For remote database access, MDARTS also supports transactions via remote procedure calls. Naturally, the response times for remote transactions are constrained by communication delays.

The remainder of this paper is organized as follows: Section 2 discusses prior work on real-time databases, Section 3 introduces the MDARTS architecture, Section 4 describes our implementation, and Section 5 states conclusions and discusses future work.

2 Background

The defining characteristic of hard real-time systems is that they must complete their computations within strict deadlines [28]. If real-time software is unable to complete a computation within its deadline, catastrophic failure may occur. These deadlines can be very tight; for example, a manufacturing machine controller may have to compute its control signals every 1 or 2 milliseconds. Failure to meet this deadline could cause the machine to become unstable and malfunction, possibly with dire consequences. Soft real-time systems, such as program trading or airline reservation systems, are sensitive to deadlines but do not fail if some are occasionally missed. MDARTS is designed for hard real-time systems.

A common misconception about real-time computing is that it is equivalent to high-speed computing [37]. Actually, there are fundamental differences between the two. Whereas high-speed computing tries to maximize average performance levels, real-time computing requires absolute performance levels. To guarantee deadlines under all conditions, hard real-time systems are designed using worst-case assumptions about all operations. Thus, real-time computing requires both high performance and predictability. Furthermore, real-time systems must keep up with external events and changes in the system being monitored. Conventional software is not constrained to respond as quickly or as predictably to external events. Because of these fundamental differences, algorithms and software architectures suitable for general-purpose computing often are inadequate for real-time systems.

Historically, real-time system designers have taken an ad-hoc approach to data management. Shared data are often simply stored in files or kept at known locations in shared memory [19]. For simple systems with little inter-task data coupling, an ad-hoc approach to data sharing suffices. However, several trends are driving future real-time systems toward complex multitasking and distributed architectures for which ad-hoc data management is inadequate. The price, performance, and reliability advantages of distributed systems encourage software designs that distribute tasks over multiple processors on a common bus or

network. Furthermore, as real-time systems become more complex, it becomes desirable to subdivide the software into separate tasks along functional lines and provide well-defined interfaces between the tasks. This functional decomposition serves to both simplify the overall design and to permit standardization and independent development of system components. When software systems are subdivided into multiple cooperating tasks, shared data must be made accessible to and used consistently by all tasks that access them. It is also necessary to control concurrent access to prevent data corruption. To solve data sharing problems consistently and efficiently, an organized approach to data management that satisfies real-time computing requirements is needed.

The Next Generation Workstation/Machine Controller (NGC) for automated factories is representative of the current trend toward distributed real-time architectures [2, 18]. The NGC is a software architecture specification for advanced machine tool controllers. It is intended to encourage standardization in the controller field and accelerate technology transfer from laboratories to industry. An NGC-compatible controller consists of multiple hardware and software components possibly supplied by different vendors. The NGC architecture is designed for high-performance real-time computing platforms such as VME-based shared-memory multiprocessors. The coordination of and communication between different software modules in the NGC architecture is accomplished via a built-in database called the Information Base Subsystem.

Because real-time systems such as the NGC require database services, several researchers have recently investigated database designs suitable for real-time applications. Most of this research addresses the performance and predictability requirements of real-time transactions. There are three primary strategies for improving the performance and/or predictability of database transactions: 1) use memory-based databases or fast database hardware, 2) schedule transactions according to task priorities, and 3) minimize delays and uncertainties associated with concurrency control and locking.

2.1 Memory-Based Databases

It is possible to dramatically enhance performance and predictability by using main memory databases that avoid disk I/O during transactions [29, 36] or by using special-purpose database machines [43]. However, main memory databases may not be feasible for some applications if the database is too large to fit into available memory or if data persistence is needed. Furthermore, it may not be desirable to keep the entire database in main memory in the first place. In general, real-time applications have heterogeneous data needs. For example, in manufacturing controllers, sensor readings are inherently volatile and need not be made persistent or recovered after failure. Other data, such as part programs and machine configuration data should be persistent but need not be updated at high speeds or frequency. Static persistent data can be retrieved from file-based databases and cached in memory for subsequent high-speed read operations. Since real-time data needs are heterogeneous, a real-time database system should provide a mix of memory- and file-based database services.

2.2 Scheduling and Priority-Based Resource Allocation

Several researchers have investigated transaction and I/O scheduling algorithms that support different real-time needs and priorities [1, 5, 24, 38, 32], and at least two commer-

cial database systems provide priority-based transaction scheduling [11, 25]. By servicing high-priority tasks first, the database can provide faster and more predictable performance for transactions submitted by high-priority tasks. In this case, low-priority tasks experience degraded performance. File-based databases in particular can benefit from transaction and I/O scheduling according to real-time priorities. Memory-based databases benefit less because the scheduling overhead may become a significant percentage of the transaction processing time. One of the difficulties with database transaction scheduling is that interactions with the operating system task scheduler must be considered.

2.3 Locking and Concurrency Control

A major source of performance uncertainty in conventional databases is the potential for transaction delays or aborts when concurrent tasks try to acquire the same locks. Unpredictable delays in transaction processing can cause critical tasks to miss deadlines. This problem and its contributing factors have been the focus of most real-time database research.

One approach to real-time database concurrency control is to abandon traditional concurrency control protocols such as two-phase locking that guarantee serializability but lead to transaction rollback delays [5, 21]. The problem with relaxing consistency constraints such as serializability is that the integrity of the data could be jeopardized. Fundamental database properties such as serializability may be called into question occasionally, but one must be careful not to lose the very properties for which the database is needed. Graham [12] claims that serializability is indispensable for correct interleavings of concurrent transaction operations. However, Lin [17, 16] suggests that, for real-time applications, data inconsistent with the external world can be worse than internally inconsistent data. He calls the correspondence between a database and the state of the external world "external consistency." External consistency is lost if the database cannot process transactions fast enough to keep up with changes in the world. Lin does not explain how a database system should choose trade-offs between internal and external consistency. Clearly, these trade-offs would depend heavily on application semantics.

Database locking in real-time systems can lead to a problem called "priority inversion." Priority inversion occurs when a high-priority task is forced to wait for a lock held by a lower-priority task. Priority inversion is sometimes unavoidable, so most researchers have investigated ways to place an upper bound on the duration of priority inversions. The most common approaches to bounding priority inversions are variants on priority inheritance protocols which temporarily boost the priority of tasks holding locks [23, 22, 27, 32]. This temporary priority boost helps tasks complete transactions and release their locks. Since the best approach to real-time concurrency control often depends upon the particular application, some researchers advocate hybrid protocols that borrow features of several earlier strategies to perform acceptably for a wider range of applications [5, 13, 14, 23, 34]. For example, Huang *et al.* [14] advocate a hybrid approach in which tasks inherit higher priorities only if they are close to committing. Otherwise, if higher-priority tasks require locks held by low-priority tasks, the lower priority transactions are aborted.

Priority inversion and locking delays can be reduced if the need for locking or the probability of locking conflicts are reduced. One approach to reducing locking conflicts is to adjust the lock granularity to lock only data that are affected by each transaction [26, 3, 35]. Another approach is to design read/write protocols that use data versioning to avoid locking altogether [40, 41, 42, 35].

3 Our Approach

MDARTS is a new approach to real-time databases. Most real-time database research has focused on specific algorithms and protocols that are compatible with real-time requirements. Very few real-time database implementations are described in the literature, and some of those seem intended primarily as testbeds for studying algorithms rather than supporting actual real-time applications [14, 33]. Besides being an implementation designed to support hard real-time applications, MDARTS improves upon prior work by:

- supporting explicit timing and semantic constraints specified by applications during application initialization,
- using object-oriented technology to customize database services at runtime,
- supporting both remote procedure call transactions and concurrent shared-memory-based transactions on multiprocessors,
- providing an extensible framework within which multiple concurrency-control protocols can coexist, and
- providing a convenient application programming interface.

In MDARTS, the database is not directly involved in scheduling task execution or attempting to meet task deadlines. These issues are considered to be beyond the scope of database responsibilities. Instead, MDARTS provides response-time guarantees for individual database transactions. These response times are specified in application code and can be used to help compute the worst-case execution time of each task. Given the worst-case execution times and the task deadlines, the application task scheduler should be able to guarantee higher-level task deadlines.

An application using MDARTS declares database variables as in the following example: DbArray<Point> **positions**("joint_positions", "read<=50usec;size=6;exclusive_update"); In this declaration, DbArray<Point> is the class name of the object **positions**, with which the application can store and retrieve elements from a shared array of Point data objects. "Point" is an application-defined data structure that contains a three-dimensional Cartesian coordinate. The two parameters within the parentheses are passed to the DbArray<Point> constructor routines which initialize the **positions** object. The first parameter is a unique identifier for that object in the database. The second parameter contains a set of semantic and timing constraints. These constraints are used during initialization to configure the database object and to verify that timing requirements will be met when transactions are performed. The following example shows how an application would perform a read transaction: Point **end_effector_position** = **positions**(5);

3.1 Object-Oriented Database Service Classes

MDARTS is based upon an object-oriented library of database service classes. Tasks needing to share data with other tasks declare objects belonging to the MDARTS database classes. These objects are automatically registered with a MDARTS Shared Data Manager (SDM) process that allocates shared memory, performs object lookup, and supports remote data access. Real-time constraints are specified by applications in the declarations of the MDARTS objects. MDARTS examines the constraints during application initialization

Constraint type	Specification
access time	"write<=80usec; read<=50usec"
persistence	"volatile"
staleness	"stale<=20msec"
concurrency	"exclusive_update"

Figure 1: Examples of real-time and semantic constraints.

and returns data handle objects that satisfy the constraints. By registering application needs during initialization, MDARTS is able to track resource allocation at runtime and guarantee response times before the actual real-time transactions are performed. Prior real-time database research either makes guarantees *a priori* with off-line static analysis of applications or makes dynamic guarantees during transaction processing [5]. Both of these approaches have significant disadvantages. Off-line analysis of transactions is not always feasible, especially for complex, distributed applications. Dynamic transaction scheduling imposes substantial overhead and cannot prevent overload conditions. MDARTS is unique in registering real-time requirements on a per-object basis during initialization. The overhead of checking these requirements and constructing the data access objects is paid only once per task before the real-time processing begins.

If the constraints cannot be satisfied, the application is notified (an exception is raised). MDARTS detects possible transaction overload conditions before the real-time processing begins. This point is crucial for safety-critical applications. Once a data handle has been constructed, the application program uses the handle to perform database transactions. The location of the data and the implementation of the data access mechanisms are hidden from applications. Some handle objects directly access shared memory for maximum transaction speeds. Some handle objects use remote procedure calls to submit transaction requests to remote MDARTS SDM servers across the network. Other handle objects access persistent data in file-based databases.

The MDARTS database library is like a population of private contractors. Each contractor is free to specialize services, such as concurrency control, to match individual application needs. Real-time and semantic constraints constitute the contracts, and each database class recognizes and implements its own set of constraints. For example, a fundamental constraint type is the response time for concurrent read or write transactions. There are many algorithms that support concurrent data access. Faster algorithms generally require semantic restrictions such as allowing only a single writer at a given time [41, 42]. If multiple concurrent writers are allowed, additional overhead is required to lock and unlock the data and to wait if another task is updating it. The MDARTS library contains data management classes optimized for restricted concurrency semantics as well as classes that support more general semantics. Each database class guarantees response times according to its own algorithm. The semantic information supplied by applications at initialization time determines which database classes will be used.

Figure 1 shows some of the MDARTS constraints and how an application can specify them using character strings. "Staleness" specifies an external consistency constraint. If a sensor monitor fails and its database values become obsolete, a staleness constraint can

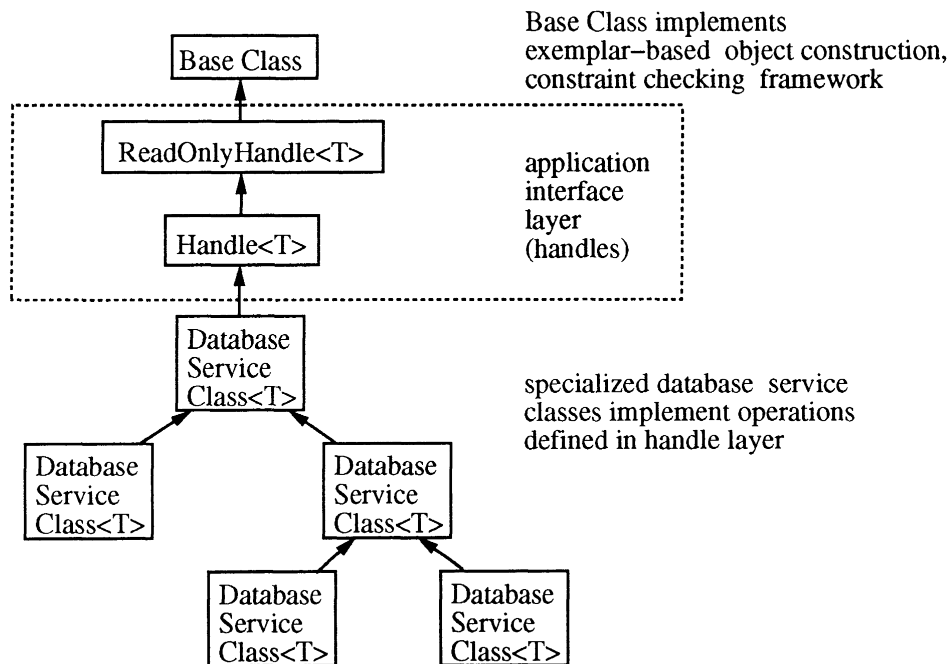

Figure 2: MDARTS database class hierarchy.

trigger an exception or warning. Several constraints can be expressed in the same character string by separating each constraint with a semicolon.

MDARTS uses exemplar-based programming to construct database objects [6]. Exemplar-based programming uses a set of prototype "exemplar" objects to instantiate objects at runtime. The exemplar objects recognize their own particular set of constraints and clone themselves if they can meet all of the constraints specified in an object construction request. This method of object construction relieves application programmers from knowing in intimate detail which database classes support which features. An application programmer chooses one of the handle classes of a MDARTS class hierarchy and specifies a set of real-time requirements for that object. The handle class defines the data access functions available to the application. Given the handle class and the constraints, MDARTS selects and instantiates an appropriate database object from the exemplar list of that handle class. Exemplar-based object construction simplifies the MDARTS application programming interface, and it makes the database library easy to extend and improve without requiring changes to application programs. Applications need only be relinked with the updated MDARTS library to use new database service objects.

Figure 2 illustrates the structure of a MDARTS class inheritance hierarchy. The leaf classes in the hierarchy, labeled "Database Service Class<T>", all provide the same data access operations but are each specialized to support different constraints. Applications access these operations through the handle interface.

It is important to determine which levels of the database class hierarchy will be visible in the application programming interface. Some object-oriented databases require applications to specify data semantics by choosing the class that supports those semantics. This approach leads to a proliferation of similar classes that the application programmer must know about. For instance, a persistent object with only one writer might be declared as

"DbPersistentExclusiveUpdateInteger my_object." In this case, applications are exposed to the leaf classes in the database library's class hierarchy. Clearly, this approach to semantic specification becomes unmanageable as the number of semantic constraints grows. In MDARTS, we pass semantic attributes like "persistent" and "exclusive_update" as strings to the exemplars rather than encoding them into the database class names. This reduces the number of different database classes to which applications are exposed and reduces dependencies between applications and the database library. Applications are only exposed to handle classes in the database hierarchy that correspond to different data interfaces (e.g., a floating point array vs. a linked list of strings). The selection of appropriate leaf classes to support the desired interface and semantic constraints is accomplished by the exemplar instantiation process.

We are using the C++ language [39] to implement our database architecture. We chose C++ because of its wide availability, runtime efficiency, compatibility with C, and object-oriented features. Some advantages of C++ for real-time software are discussed in the literature [8, 7]. To enhance the portability of our implementation, we are using only standard features of C++ rather than adding language extensions.

3.2 Shared-Memory Objects

The timing constraints of some real-time applications are such that database transactions on the order of tens of microseconds may be needed. For example, an NGC manufacturing machine controller will typically have one or more tasks monitoring sensors and one task computing control signals. The control task is periodic with a hard deadline every millisecond. Each time the control task runs, it extracts the current sensor values from the database and computes new control signals for the machine actuators. To guarantee the control task deadline, the worst-case database response times to access each shared datum must be much less than one millisecond.

Disk-based databases, or virtual memory-based databases that may generate page faults, cannot yet approach this speed (recall that we are talking about worst-case access time). Therefore, data accessed at extremely high speeds must be kept in shared physical memory. Because of address space differences across multiple tasks or processors, shared memory cannot be used as easily as local process memory for instantiating C++ objects. Specifically, pointers to data or to functions cannot be shared easily across different processes. In C++, objects often contain pointers to functions (most often in the form of a pointer to a virtual function table). Jordan [15] discusses this problem and presents an approach to instantiating C++ objects in shared memory. Unfortunately, Jordan's methods rely on virtual memory and will not work for real-time operating systems, such as VME-based VxWorks, that do not support virtual memory. Therefore, we restrict the C++ objects that are kept in shared memory so that they do not use the virtual function mechanism. MDARTS splits its objects into local handles with virtual functions that reside in each application process and shared data objects that reside in shared memory. This permits the portable use of C++ object-oriented features together with shared memory. Multiple MDARTS handle objects corresponding to the same database object may exist in a distributed application, and each will store and retrieve data from the same shared-memory location.

Most of the algorithms proposed for real-time databases assume a client-server architecture in which a database process services multiple application tasks. In this context, the scheduling of transaction processing in the server must be explicitly coordinated with

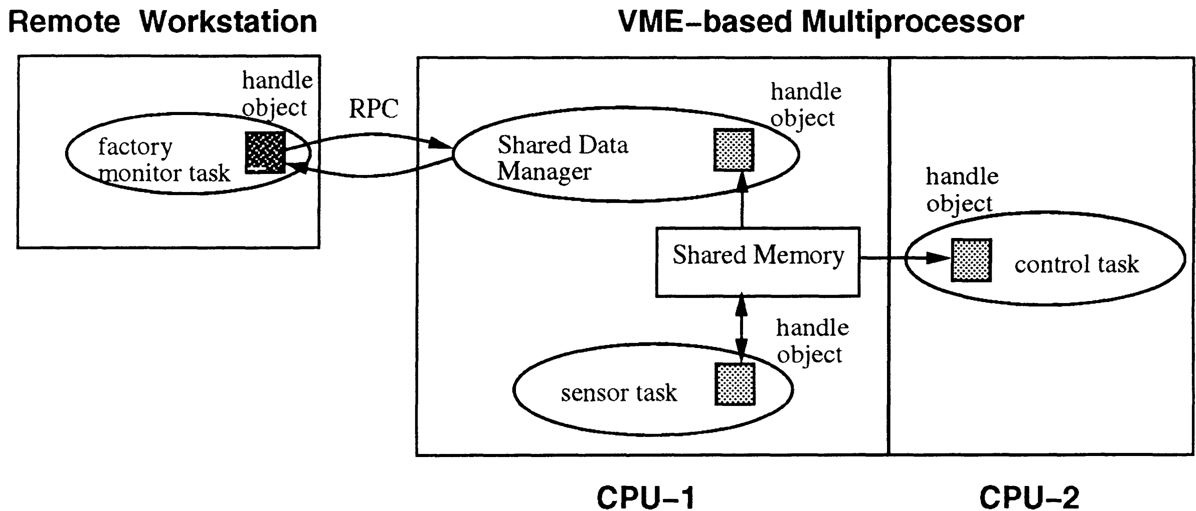

Figure 3: Access to shared memory data.

the scheduling of application tasks. Even if an ideal transaction scheduling algorithm were available so that the server could instantly choose the best action, the client-server architecture has fundamental limitations for high-speed applications. These limitations are the communication latency and the context switching overhead implicit in client-server interactions. MDARTS avoids these problems by moving transaction processing for shared-memory data into the application tasks. Transaction scheduling is implicitly accomplished by the application task scheduler.

For maximum performance, it is necessary for transactions to directly access data in shared memory without communicating with a separate database server process. However, without a database process to manage concurrency, the database objects must supply their own concurrency control. In other words, the database objects should be atomic data types [44, 31]. An atomic data type is essentially a class whose member functions guarantee serial behavior in the presence of concurrent requests. Since the concurrency control can be individually tailored according to the semantics of the class member functions, it is possible to achieve higher levels of concurrency than with traditional read-write locking [30, 45]. It is also possible to implement atomicity in a base class and inherit this property in derived subclasses [9]. MDARTS shared-memory objects differ from atomic data types described in the literature in that the handle parts of the objects are fragmented across multiple separate processes. The data itself and lock information needed to synchronize access is kept in shared memory to ensure its consistency across all processes.

3.3 RPC

In some cases, it is impossible to directly access the shared memory where data reside because the processor does not share the same bus. For remote access to data, MDARTS supports a remote procedure call (RPC) interface. Each database object is registered with the SDM process that runs as an RPC server on the workstation where the object resides (see Figure 3). Applications either access the shared memory directly with their database handles, or their database handles use remote procedure calls to ask the SDM to perform transactions on their behalf. Applications need not know whether the access is local or

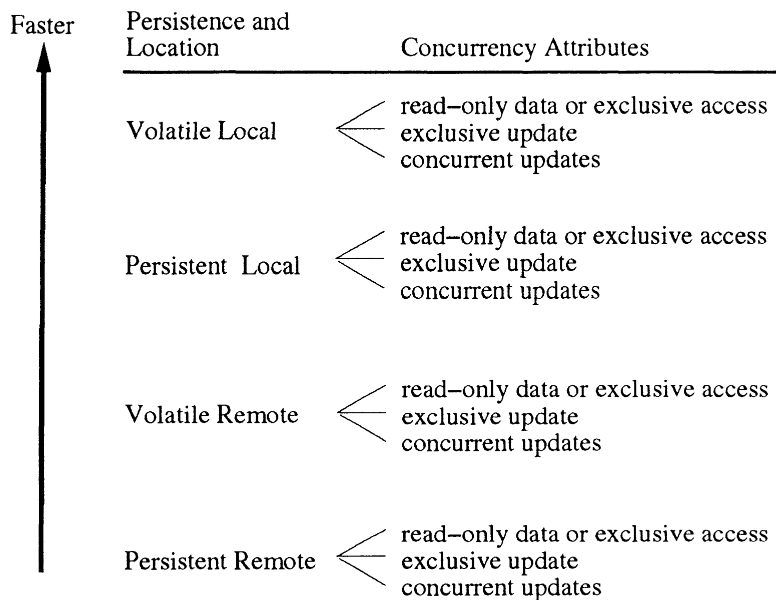

Figure 4: A possible performance spectrum for a heterogeneous database.

remote; the database handles hide the access mechanism. MDARTS takes communication delays into account when providing transaction time guarantees for remote access. The SDM keeps track of the location and identity of shared-memory objects and also constructs its own database handle for each object to service remote requests.

For RPCs, we are using the Network Computing System (NCS) RPC library from Hewlett Packard [10]. NCS was chosen as the core technology for the OSF Distributed Computing Environment (DCE). Some advantages of NCS over Sun RPC are described in [20]. Although we use NCS, our architecture is not dependent upon NCS in any fundamental way, and another RPC mechanism could be substituted.

3.4 Heterogeneous Databases

Most previous work on real-time databases assumes a homogeneous architecture for the database management system. That is, the database is either file-based or memory-based, local or distributed. A common problem with using a homogeneous database architecture is that data with fast (volatile or local) semantics end up being managed by a database system designed for data with slower (persistent or distributed) semantics.

Figure 4 illustrates a performance spectrum corresponding to variations in data persistence, location, and transaction concurrency (the performance spectrum of a given system may differ from Figure 4 according to variations in network, CPU, and disk speeds). Worst-case transaction response times could easily vary by five orders of magnitude from the slowest to the fastest combination shown in Figure 4. In many cases, it is possible to dramatically improve database performance by taking these semantic differences into account. If persistence is unnecessary for a data object (such as a periodic sensor reading), transactions accessing that object can be processed without disk I/O. This means the database system will be able to provide much faster response times. Furthermore, if only one process will be updating a data object, concurrency control can be simplified, and trans-

action times for updates can be made more predictable. Data in real-time systems often exhibit such favorable semantics.

3.5 External Database Interfaces

MDARTS provides external database interfaces for accessing other database systems and for MDARTS data objects that must be persistent. A machine controller in a factory may need to update an external factory-wide database to periodically report its activities. This external database will likely be a file-based, non-real-time database system. Clearly, the controller will have to perform such updates outside of any fast feedback loops. MDARTS can construct handles to external databases that hide the database-specific queries from application programs. These handles will not guarantee fast response times, but they will provide the same convenient application programming interface as other MDARTS objects.

A common approach to providing persistence for object-oriented applications is to delegate storage management to a file-based database management system. In his book on object-oriented design [4], Booch shows how C++ can be used to access a relational database system. The OZ+ object-oriented database management system [46] uses an underlying relational database to supply object persistence. Likewise, MDARTS can use file-based database management systems for objects that must be persistent. As with external database interfaces, these persistent handles cannot support the transaction speeds needed for high-speed control loops. However, not all of the data needed by real-time applications must be accessed at high speeds. For example, a file-based database could be used to store static configuration data such as machine specifications. In some cases, it will be possible to cache this data in memory so high-speed read operations may be supported.

4 Implementation of Real-Time Constraints

4.1 Constraint Specification

Prior real-time database research investigates algorithms suitable for meeting real-time requirements. However, little attention has been paid to the question of how real-time applications should communicate their requirements to the database system. This issue is very important in any practical implementation. A common assumption is that queries should contain deadline information and/or task priorities. The problem with putting deadline requirements in queries is that if the database system cannot meet a deadline, this will not be discovered until the query is performed. Furthermore, processing deadline information adds additional overhead and complexity to query processing. This overhead in turn reduces the performance of the database system. In MDARTS, deadline information is processed during database handle initialization so errors are detected early, and overhead during transaction processing is minimized.

Real-time constraint specification is important because real-time applications often contain implicit assumptions about the performance characteristics of data access operations. For example, the control loop of a machine controller might access a database to read sensor values or issue actuator commands. The sampling frequency of the control loop is determined *a priori* according to the control strategy and the physical characteristics of the machine. The sampling frequency helps determine the control task's deadline. To verify

```
/***********************************************************
* Declaration of MDARTS variable in sensor task that will be updating it:
*/

static  DbArray<Point> sensor_array("position_sensors",
   "exclusive_update; size = 6; access_time <= 50usec",  CREATE);

/*  Sensor task updates the data:
*/

sensor_array(5)  =  Point(1.2, 0.866, 3.4);

/***********************************************************
* Corresponding declaration of MDARTS variable in control task:
*/

static ReadOnlyDbArray<Point> position_sensor_array("position_sensors",
   "read <= 80usec");

/*  Control task reads the data:
*/

int  i  =  position_sensor_array.size() – 1;

Point  end_effector_position  =  position_sensor_array(i);

/***********************************************************/
```

Figure 5: MDARTS C++ application programming interface.

that the deadline will be met, the software designer must know the time required to complete the database transactions. However, if this timing dependency is implicit and is not checked at runtime, the application may fail catastrophically if the designer's assumptions are wrong or the database implementation is changed. Therefore, applications should specify their real-time constraints explicitly and delegate the responsibility for meeting them to the database system. The application creates a contract, and MDARTS either accepts or rejects the terms of the contract. In this way, subtle errors are avoided, and application code is decoupled from the database implementation.

Data semantics and real-time constraints should be explicitly specified in application code near the point of use. In MDARTS, this information is passed to the object constructor functions. Figure 5 illustrates the MDARTS C++ application programming interface (API). The C++ "static" keyword indicates that these handle objects are constructed at program initialization time. Notice that the syntax for accessing the MDARTS database is as convenient and natural as that of ordinary C++ objects. Furthermore, unlike preprocessor-based application interfaces such as embedded SQL, MDARTS handles are first-class objects that can be dynamically created and passed as parameters to subroutines.

The MDARTS handle classes in Figure 5 are DbArray<T> and ReadOnlyDbArray<T>, where <T> indicates a template class that is parameterized by class T. In this case, T is an application-defined class called "Point," which represents a 3-dimensional Cartesian coordinate. An array of points might be used to store the positions of each joint in a robot arm. The same MDARTS template classes that manage arrays of Point objects in Figure 5 can also manage arrays of other types of data objects. Thus, with template instantiation,

new data structures designed by application programmers can be added to the MDARTS database library very easily.

The "exclusive_update" constraint specified by the sensor task in Figure 5 causes MDARTS to reject subsequent attempts to construct handle objects that could modify the data. This constraint allows the DbArray<T> class to use efficient concurrency control algorithms and provides protection from unauthorized data access. By alternating updates to two copies of the data as described by Vidyasankar [41], MDARTS can perform concurrent read and write transactions without locking the data. This technique relies on the restriction that only one write transaction will be active at a given time. The "exclusive_update" constraint guarantees that this will be the case. It is important to note that constraints such as "exclusive_update" are checked only during initialization of the data handles. Subsequent database access using the handles is not burdened with the overhead of checking access permissions. In Figure 5, the control task declaration specifies its handle object as a ReadOnlyDbArray<Point>. This class cannot update the data, so it satisfies the "exclusive_update" constraint. If the application programmer mistakenly tries to modify data with a ReadOnly handle, an error is reported at compile time. We implement the ReadOnly semantic restriction as a separate handle class so that such errors can be detected by the compiler. This is an exception to our rule of keeping semantic constraints out of the handle class names. If "read-only" were only specified in the constraint string, illegal update attempts would not be caught until runtime.

MDARTS provides a rich environment for developing real-time database constraints. For example, researchers interested in new atomic data type concurrency control techniques can implement their algorithms in database service classes within the MDARTS framework by deriving new classes from existing MDARTS classes. The exemplar-based object construction functions are inherited from the base class, so researchers can focus on implementing the new constraints, paying minimal attention to the MDARTS framework. Since constraints are expressed as character strings by applications, it is possible for researchers to invent a new semantic constraint, define a syntax for the constraint, and integrate it seamlessly with MDARTS. It is not even necessary to recompile the MDARTS library to add a new service class or a new constraint. User-defined MDARTS classes can simply be linked into the application code along with the standard MDARTS library. Exemplars that do not recognize the new constraint will simply reject the handle construction request and pass the request on to one of the user-defined exemplars that recognizes it.

4.2 Constraint Implementation

Specification of constraints is useful only if the database system can meet the constraints. Therefore, the implementation of constraints is a crucial part of MDARTS. Rather than establishing a single policy or protocol for deadline guarantees, MDARTS provides a framework within which many approaches to providing deadline guarantees can coexist. As we populate the MDARTS database service library, we are developing various constraint implementation strategies. Since these strategies are encapsulated in the object implementations, different strategies can also be added to the MDARTS library by adding new objects that use different algorithms.

The MDARTS constraint checking method requires that database class exemplars know their own performance characteristics and reflect them appropriately as handle objects are constructed. Performance is inherently implementation-dependent, so it is appropriate that

implementations of database objects guarantee their own performance. It is the responsibility of the database class implementor to ensure that timing constraints are correctly implemented by the database classes.

The primary goal of MDARTS is to provide fast, predictable response times for data access. Therefore, our main implementation focus is on timing constraints. To guarantee fast response times, we have followed several principles:

- **When possible, use shared memory.** Shared memory provides the highest performance for systems that support it.
- **Support distributed, concurrent transaction processing.** Centralized transaction execution or lock management can become a bottleneck in multiprocessor systems.
- **Avoid unnecessary locking.** When possible, use data versioning [35] or multiple data copies [41] to permit concurrent read and write operations without locking.
- **Match locking granularity with data semantics.** This ensures that locking does not unnecessarily restrict concurrency. Sha *et al.* [26], Badrinath and Ramamritham [3], and Son [35] each propose locking only the data affected by a transaction. However, identifying affected data and locking only those data are non-trivial problems in conventional database systems, where the data affected by a transaction are determined during query processing at runtime. MDARTS simplifies this problem since semantically-related data are grouped into database objects. Each object supports transactions on its data, so it is easy to match locking granularity with transaction semantics.
- **Control locking duration.** Locks should be acquired and released within the critical sections of transaction operations performed by database service objects. MDARTS objects can provide operations with greater semantic content than simple read or write transactions. For example, a counter object can provide increment and decrement operations which lock the data only long enough to perform the update. By encapsulating the lock and unlock operations in the database operation, MDARTS can guarantee short critical sections.
- **Prevent unconstrained priority inversion.** Priority inversion can be limited by disabling task preemption during short transaction critical sections. For transactions with long critical sections, it may be possible to avoid priority inversion by avoiding locking with data versioning techniques.

Using these principles, we have created database service classes that provide deadline guarantees. In our implementation, MDARTS database objects that use shared memory and require no locking typically guarantee simple read/write transaction response times of about twenty microseconds (using 25 MHz 68030 processors on a VME-based multiprocessor running the VxWorks operating system). If application semantics require multiple concurrent updates, we have implemented a first-come-first-served locking protocol. This protocol uses the test-and-set instruction for mutual exclusion and disables preemptions during critical sections. It guarantees response times of $(n * c) + t$, where there are n processors running tasks interested in updating the locked data; c is the time required to execute the critical section plus some overhead associated with checking the lock; and t is transaction execution time outside of the critical section. With preemption disabled, the database response time is independent of how many updating tasks run on each processor since at most one of them can request a lock at any given time.

Remote transactions executed via RPC are much slower than transactions that use direct shared memory access. The same multiprocessor system that yields 20 microsecond response times for shared-memory transactions typically requires 16 milliseconds to perform a remote transaction. Almost all of this time is consumed in the RPC overhead required to send messages across the Ethernet (or the VME backplane, for socket-based communication between multiprocessor CPUs). VxWorks supports local message passing services that are faster than the socket communications used by RPC, but tasks on local CPUs should directly access shared memory in the first place.

It is especially important for remote transactions that the locking and unlocking of data is accomplished within the database operation's critical section rather than under control of the application. Otherwise, locks held across the network by remote applications could block execution of local transactions for extended periods of time, making it impossible to guarantee fast response times for any of the tasks that access the data.

The object-oriented architecture of MDARTS has been crucial to the success of the real-time constraint implementation effort. Each database class is individually designed with its own transaction operations and concurrency control. This results in an extensible system that avoids bottlenecks and allows each database class to implement its own performance guarantees. During initialization, each task predeclares its intention to update MDARTS objects, and each MDARTS object keeps track of the tasks that intend to perform transactions on it. With this information and the object's own knowledge of transaction execution time, MDARTS can make performance guarantees.

During initialization, MDARTS classes adjust their performance estimates according to the actual performance of the underlying network and computing platform. To determine the platform's performance, MDARTS can time the execution of a known suite of subroutines. Alternatively, performance information can be read from a file maintained by the application programmer. In either case, MDARTS exemplar objects use this platform-specific data to derive estimates of their own performance for various operations. This runtime self evaluation by the database provides flexibility in configuring and upgrading the database without requiring exhaustive testing and configuration analysis.

5 Conclusion

In this paper, we have presented a new object-oriented architecture for real-time databases. MDARTS is designed to support hard real-time systems such as manufacturing machine tool controllers. In our implementation, we have demonstrated guaranteed response times of tens of microseconds on shared-memory multiprocessors, which are commonly used for this class of applications. MDARTS is opportunistic; it can specialize its services and algorithms according to semantic information provided by applications at runtime.

Currently, we have completed implementation of the MDARTS framework with the Shared Data Manager, local and remote updates, and exemplar-based object construction using semantic and timing constraints. We have also implemented shared lock objects for concurrency control. MDARTS currently runs on VxWorks-based shared-memory multiprocessors and on Sun workstations using System V shared memory and socket communication. Future work includes providing file-based database interfaces, extending MDARTS to support event-driven application callbacks, and demonstrating MDARTS by integrating it with an actual machine tool controller.

Acknowledgement

The authors would like to thank Elke Rundensteiner for her feedback and helpful suggestions on an earlier draft of this paper.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions," *SIGMOD Record*, vol. 17, no. 1, pp. 71–81, March 1988.

[2] B. Anderson, "Next generation workstation/machine controller (NGC)," in *Proc. IPC '92*, pp. xix–xxvi, April 1992.

[3] B. R. Badrinath and K. Ramamritham, "Synchronizing transactions on objects," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 541–547, May 1988.

[4] G. Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., 1991.

[5] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal, "Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control," in *Proc. IEEE Int'l Conf. on Data Engineering*, pp. 470–480, February 1989.

[6] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison Wesley, 1992.

[7] I. J. Cox, "C++ language support for guaranteed initialization, safe termination and error recovery in robotics," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, volume 1, pp. 641–643, 1988.

[8] I. J. Cox, D. A. Kapilow, W. J. Kropfl, and J. E. Shopiro, "Real-time software for robotics," *AT&T Technical Journal*, vol. 67, no. 2, pp. 61–71, March/April 1988.

[9] D. Detlefs, M. Herlihy, and J. Wing, "Inheritance of synchronization and recovery properties in Avalon/C++," *IEEE Computer*, vol. 21, no. 12, pp. 57–69, December 1988.

[10] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant, "The network computing architecture and system: An environment for developing distributed applications," in *Proc. Summer USENIX Conference*, pp. 385–398, June 1987.

[11] GDX. sales literature of Firmware Associates, Inc., West Chester, PA, 1992.

[12] M. H. Graham, "Issues in real-time data management," *Journal of Real-Time Systems*, vol. 4, no. 3, pp. 185–202, September 1992.

[13] J. R. Haritsa, M. J. Carey, and M. Livny, "Data access scheduling in firm real-time database systems," *Journal of Real-Time Systems*, vol. 4, no. 3, pp. 203–241, September 1992.

[14] J. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley, and B. Purimetla, "Priority inheritance in soft real-time databases," *Journal of Real-Time Systems*, vol. 4, no. 3, pp. 243–268, September 1992.

[15] D. Jordan, "Instantiation of C++ objects in shared memory," *Journal of Object-Oriented Programming*, pp. 21–28, March/April 1991.

[16] K.-J. Lin, "Consistency issues in real-time database systems," in *22nd Hawaii Int'l Conf. on System Sciences*, January 1989. Available in RTCL library.

[17] K.-J. Lin and M.-J. Lin, "Enhancing availability in distributed real-time databases," *SIGMOD Record*, vol. 17, no. 1, pp. 34–43, March 1988.

[18] *Next Generation Workstation / Machine Controller Specification for an Open System Architecture Standard*, Martin Marietta Astronautics Group, NGC-0001-13-000-SYS edition, March 1992.

[19] J. Meisenbacher, "RTOM: a real-time DBMS concept," in *Proc. IEEE National Aerospace and Electronics Conference*, pp. 269–274. IEEE, May 1991.

[20] N. W. Mishkin, "Apollo NCA and Sun ONC: A comparison," Posted by the author to USENET, September 1989.

[21] S. Nishio, S. Taniguchi, and T. Ibaraki, "On the efficiency of cautious schedulers for database concurrency control – why insist on two-phase locking?," *Journal of Real-Time Systems*, vol. 1, pp. 177–195, 1989.

[22] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 116–123, 1990.

[23] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proc. Real-Time Systems Symposium*, pp. 259–269, December 1988.

[24] K. Ramamritham, "Real-time databases," *Int'l Journal of Distributed and Parallel Databases*, 1992. Invited Paper - to be published.

[25] *RTA Introduction & Overview*, Real Time Computersoftware Ges.m.b.H., 1992.

[26] L. Sha, J. P. Lehoczky, and E. D. Jensen, "Modular concurrency control and failure recovery," *IEEE Trans. Computers*, vol. 37, no. 2, pp. 146–159, February 1988.

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency control for distributed real-time databases," *SIGMOD Record*, vol. 17, no. 1, pp. 82–98, March 1988.

[28] K. G. Shin, "Introduction to special issue on real-time systems," *IEEE Trans. Computers*, vol. 36, no. 8, pp. 901–902, August 1987.

[29] M. Singhal, "Issues and approaches to design of real-time database systems," *SIGMOD Record*, vol. 17, no. 1, pp. 19–33, March 1988.

[30] A. H. Skarra, "Concurrency control for cooperating transactions in an object-oriented database," *SIGPLAN Notices*, vol. 24, no. 4, pp. 145–147, April 1989.

[31] A. H. Skarra and S. B. Zdonik, "Concurrency control and object-oriented databases," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, editors, pp. 395–421. Addison Wesley, 1989.

[32] S. H. Son, "Scheduling real-time transactions," in *Proc. EuroMicro '90 Workshop on Real Time*, pp. 25–32. IEEE, 1990.

[33] S. H. Son and Y. Kim, "A software prototyping environment and its use in developing a multiversion distributed database system," *International Conference on Parallel Processing*, vol. 2, pp. 81–88, August 1989.

[34] S. H. Son, J. Lee, and Y. Lin, "Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control," *Journal of Real-Time Systems*, vol. 4, no. 3, pp. 269–276, September 1992.

[35] S. H. Son, "Semantic information and consistency in distributed realtime systems," *Information and Software Technology*, vol. 30, no. 7, pp. 443–449, September 1988.

[36] S. H. Son, "Recovery in main memory database systems for engineering design applications," *Information and Software Technology*, vol. 31, no. 2, pp. 85–90, March 1989.

[37] J. A. Stankovic, "Misconceptions about real-time computing," Technical report, University of Massachusetts, Amherst, 1988. Available in RTCL library.

[38] J. A. Stankovic and W. Zhao, "On real-time transactions," *SIGMOD Record*, vol. 17, no. 1, pp. 4–18, March 1988.

[39] B. Stroustrup, *The C++ Programming Language Second Edition*, Addison Wesley, 1991.

[40] P. Tang, P.-C. Yew, and C.-Q. Zhu, "A parallel linked list for shared-memory multiprocessors," in *IEEE Int'l Computer Software & Applications Conf.*, pp. 130–135, September 1989.

[41] K. Vidyasankar, "An elegant 1-writer multireader multivalued atomic register," *Information Processing Letters*, pp. 221–223, March 1989.

[42] K. Vidyasankar, "Concurrent reading while writing revisited," *Distributed Computing*, pp. 81–85, 1990.

[43] G. von Bultzingsloewen, K. R. Dittrich, C. Iochpe, R.-P. Liedtke, P. C. Lockemann, and M. Schryro, "KARDAMOM – a dataflow database machine for real-time applications," *SIGMOD Record*, vol. 17, no. 1, pp. 44–50, March 1988.

[44] W. Weihl and B. Liskov, "Implementation of resilient, atomic data types," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 2, pp. 245–269, April 1985.

[45] W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1488–1505, December 1988.

[46] S. P. Weiser and F. H. Lochovsky, "OZ+: An object-oriented database system," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, editors, pp. 309–337. Addison Wesley, 1989.