MDARTS: A Multiprocessor Database Architecture for Hard Real-Time Systems

Victor B. Lortz Kang G. Shin Jinho Kim

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

{vbl,kgshin,jhokim}@eecs.umich.edu
(734) 763-0391 (voice); (734) 763-4617 (fax)
October 12, 1998

ABSTRACT

Complex real-time systems need databases to support concurrent data access and provide well-defined interfaces between software modules. However, conventional database systems and prior real-time database systems do not provide the performance or predictability needed by high-speed, hard real-time applications. To address this need, we have designed, implemented, and evaluated an object-oriented database system called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). MDARTS avoids the client-server overhead of most prior real-time database systems by moving transaction execution into application tasks. By eliminating these sources of overhead and focusing on basic data management services for control systems (data sharing, serializable transactions, and multiprocessor support), our MDARTS prototype provides hard real-time transaction times approximately three orders of magnitude faster than prior real-time database systems. MDARTS ensures bounded locking delay by disabling preemption when a transaction is waiting for a lock, and hence allows for the estimation of worst-case transaction execution times.

Another contribution of MDARTS is that it supports explicit declarations of real-time requirements and semantic constraints within application code. The MDARTS library examines these declarations at application initialization time and attempts to construct objects that are compatible with the requirements. Besides local shared-memory transactions with hard real-time response time guarantees, MDARTS also supports remote transactions that use remote procedure calls for data access with less stringent timing constraints. Our MDARTS prototype is implemented in C++, and it runs on VME-based multiprocessors and Sun workstations.

The work reported in this paper was supported in part by the National Science Foundation under Grants DDM-9313222 and IRI-9504412.

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

1 Introduction

Real-time systems are an increasingly important class of computer applications. Examples of real-time systems include advanced manufacturing systems, air traffic control systems, telecommunications systems, nuclear reactor controllers, and "smart" weapons systems. A computation is considered real-time if its correctness depends in part on the time at which it completes. In other words, the computations have deadlines associated with them. Real-time systems can be categorized as either hard or soft real-time. In a soft real-time system, the value of the computations is sensitive to deadlines, but the system will not fail if some deadlines are occasionally missed. In hard real-time systems, catastrophic failure can occur even if one deadline is missed. Although the techniques presented in this paper could be used for soft real-time systems, we focus primarily on hard real-time systems.

As real-time applications become more complex or need to process large volumes of data, it becomes desirable to use database systems to manage data shared between software components (tasks, processes, modules). For example, in a manufacturing system, a database can be used to store part specifications, part programs, machine characteristics, control equation parameters, histories of performance data, and the current state of the machine(s). If this information is available in a database, it can be used to support both low-level servo control and high-level supervisory control of manufacturing machines. Furthermore, it becomes much easier to integrate new sensors and software modules into the controller because their interactions with other parts of the controller can be defined in terms of operations on the database. The primary difficulty in using databases in real-time systems is that most database systems are not designed to provide the performance levels or real-time guarantees needed by high-speed real-time systems. High-speed is a relative term. We consider a real-time system to be high-speed if it typically requires worst-case database transaction times of less than a millisecond. This definition of high-speed is somewhat arbitrary, but it is motivated by the hard deadline constraints of machine tool controllers that have control tasks with periods of about one millisecond.

It is possible to improve database performance by keeping the database in memory and avoiding disk I/O during transaction processing [11]. However, conventional main memory databases are designed to maximize average throughput, not to minimize individual transaction times. Typical average response times for simple transactions in main memory databases (600 milliseconds for TPK [25], about 69 milliseconds for the main memory version of Starburst with concurrency control disabled [22], over 100 milliseconds for PRISMA/DB [3]) are much too slow for high-speed hard real-time systems. Furthermore, these main memory database systems do not provide worst-case guarantees for their transactions. Hard real-time systems need worst-case guarantees to ensure that all deadlines will be met.

Of the prior real-time database prototypes or simulations reported in [2, 5, 7, 21, 23, 14, 36, 40, 47, 51], none provides hard real-time guarantees, and none has average transaction times of database systems are not suitable for high-speed hard real-time systems such as machine tool controllers. Because suitable database management systems have been unavailable, hard real-time systems have traditionally used ad hoc methods for data management. However, ad hoc methods do not provide the flexibility needed for the complex, evolving software architectures of next-generation hard real-time systems. To provide greater flexibility and to manage the complexity of future hard real-time applications, better real-time data management technology is needed.

To meet this need, we have designed and implemented a hard real-time database system called MDARTS (*Multiprocessor Database Architecture for Real-Time Systems*). MDARTS is a framework for developing object-oriented data management services suitable for high-speed hard real-time applications on uniprocessor or multiprocessor computing platforms. Our MDARTS prototype is an extensible library of data management classes written in C++ [45]. The MDARTS library doesn't use a client-server architecture but permits application tasks to access directly databases in shared memory. By eliminating the IPC (Inter Process Communication) overhead of client-server architecture, MDARTS can provide high-speed real-time transaction response times (of microsecond order) required in, for example, manufacturing applications. Applications using MDARTS can specify hard real-time transaction response requirements in the declarations of their database objects. MDARTS checks, prior to executing transactions, if these hard real-time requirements can be met or not in order to provide a hard real-time guarantee for every transaction.

MDARTS is not intended to duplicate the services of a traditional database system, since many of these features are expensive to provide and are not necessary in the context of most hard real-time systems [26, 37]. For example, most database systems provide interpreters for ad hoc queries expressed in database languages such as SQL. In hard real-time application domains, the raw performance of the database is much more important than user-friendly interfaces. Furthermore, some of transaction management features (e.g., concurrency control and recovery mechanisms) supported by conventional database systems can be too expensive for some high-speed real-time systems. Thus, MDARTS doesn't require that all transactions preserve a single set of transaction properties (i.e., all ACID properties). Instead, MDARTS permits transactions to choose their own implementation of transaction management features suitable for applications. MDARTS does address an important problem domain that has not been adequately addressed before. Specifically, MDARTS provides flexible data management services that are compatible with the extremely demanding performance requirements of high-speed hard real-time systems. To achieve this goal, MDARTS employs a new transaction model registering transaction properties on a per-object basis. MDARTS also provides a new concurrency control mechanism that bounds locking delay by disabling preemption of each transaction waiting for a lock. With these features, MDARTS can estimate the worst-case transaction time by using the execution time of each transaction calibrated automatically in advance and its bounded locking delay. Our prototype MDARTS has been implemented on a shared-memory multiprocessor and a commercial real-time operating system kernel (20 MHz 68030 processors running VxWorks). On this platform, MDARTS can guarantee transaction times of less than 100 microseconds for simple local transactions typical of machine tool controllers. Local transactions access directly databases in shared memory through the multiprocessor bus (i.e., VME bus), which can provide hard real-time predictability by eliminating IPC overhead of client-server architecture. Moreover, MDARTS also supports remote transactions over a network via remote procedure calls (RPC). Due to the less predictable latency of RPC, remote transactions can't provide hard real-time guarantees but these can be used to support remote users accessing database objects across a network. These relatively slow RPC transactions do not delay the fast local transactions executed by application tasks on the multiprocessor. Except for variations in transaction time guarantees, the locations and implementations of MDARTS objects are transparent to applications.

The remainder of this paper is organized as follows. In Section 2 reviews real-time scheduling theory and discuss prior real-time database work. Section 3 discusses the MDARTS transaction model. Section 4 presents the MDARTS architecture and discusses implementation issues. Section 5 describes the techniques used to automatically calibrate MDARTS database objects on a computing platform. Section 6 describes our implementation platform and strategy for conducting performance evaluation of our MDARTS prototype. Section 7 presents the timing results of our experiments and discusses the worst-case transaction times. The paper concludes with Section 8.

2 Background

The fundamental goal of a real-time database system is to facilitate the development of real-time applications. When developing a real-time application, particularly a hard real-time application, it is necessary to analyze the tasks that comprise the application to verify that all (or as many as possible) task deadlines will be met. Verifying that task deadlines will be met is called *schedulability analysis*, and this problem has been extensively studied in the literature. Rate monotonic scheduling is an optimal algorithm for static (fixed) task priorities; if any static priorities can meet all the deadlines for a given task set, then so can rate monotonic priorities [27, 32]. Various dynamic priority scheduling protocols have also been studied (earliest due date, least slack time, etc. [8]).

To make schedulability guarantees in a hard real-time scheduling algorithm, it is necessary to know in advance the computation times, periods, and blocking times of all application tasks. For example, the rate monotonic scheduling algorithm defines a set of inequalities (Equation (2.1)) that, when satisfied, guarantee the schedulability of a set of n tasks. Here C_i, P_i, and B_i represent the computation time, period, and blocking time of task τ_i, respectively, and these all must be determined in advance.

$$\forall i, 1 \leq i \leq n \quad \frac{C_1}{P_1} + \frac{C_2}{P_2} + \cdots + \frac{C_i}{P_i} + \frac{B_i}{P_i} \leq i(2^{1/i} - 1) \tag{2.1}$$

However, the worst-case computation times of tasks are sometimes difficult to determine. Furthermore, unless resource-sharing protocols can bound the blocking time B_i, it is impossible to guarantee task deadlines. When a database transaction is performed by a real-time task, the delays and execution time uncertainties associated with concurrency control and recovery can make schedulability analysis impossible. MDARTS addresses this problem by making transaction times short and predictable, and hence can support hard real-time schedulability.

We characterize the real-time performance of MDARTS with the term *transaction time*. An MDARTS transaction time consists of two components. The first component represents the worst-case execution time C_t required to perform transaction t. The second component represents worst-case blocking delays B_t caused by transaction t either for concurrency control, I/O, or communication with other processes. The two components of an MDARTS transaction time correspond to the parameters needed to perform real-time schedulability analysis of tasks that use MDARTS. For simplicity, we sometimes refer to a transaction time as a single number. In those cases, we mean the sum of the two components.

2.1 Related Work

2.1.1 Real-Time Database Systems

Several researchers have recently investigated real-time database systems (RTDBSs). For a database system to be suitable for a real-time system, it must have fast and predictable transaction times. In other words, the C_t and B_t components of the database transactions must be small and bounded as tightly as possible so that the inequalities of Equation (2.1) are satisfied. Prior RTDBS research investigates three primary strategies for improving the performance and predictability of database transactions:

1. use memory-based databases [9, 11, 37, 41],
2. schedule transactions according to task priorities or deadlines [1, 6, 33, 43, 42],
3. reduce delays and uncertainties associated with concurrency control [6, 9, 13, 15, 20, 30, 31, 35, 36, 39, 48].

To this list, we add:

4. avoid the overhead associated with a client-server architecture, and
5. run transactions in parallel on shared-memory multiprocessors.

Some real-time database systems have been developed for main memory which can dramatically enhance performance and predictability by avoiding disk I/O delays [37, 40]. Garcia-Molina and Salem presented a nice overview of main memory database research in [11]. STRIP (STanford Real-time Information Processor) [2] and StarBase [23] are the examples of main-memory real-time database systems. These prior systems use client-server architectures thus incurring significant IPC overhead. Thus, they can't provide high-speed (of microsecond order) real-time response and a hard real-time guarantee for each and every transaction.

More recently, Dali [17], RTSORAC [51], and RTDM (Real-time Data Manager) [5] have been developed as main-memory object-oriented database systems. Like MDARTS, they all maintain databases in shared memory and don't use a client-server architecture in order to provide high performance required in real-time systems. Dali [17] is a main memory storage manager incorporating two phase locking protocol and a new elegant mechanism for write-ahead logging. In addition to the unpredictability caused by the recovery manager that invokes disk I/O's to write log records, the two phase locking protocol can cause unbounded delays for transactions to acquire a lock.

The RTSORAC system [51] has been developed using the object-oriented model, RTSORAC, for soft real-time databases. The model representing objects, relationships, and transactions has constructs similar to those in MDARTS. But the semantic locking concurrency control of the RTSORAC system cannot guarantee bounded locking delays. RTDM [5] is a main-memory object-oriented database system supporting time-constrained transactions for real-time command and control systems. Even though RTDM employs a new concurrency control algorithm extending Priority Ceiling Protocol [32], it doesn't prevent transactions from uncertain blocking in acquiring a lock. By contrast, MDARTS proposes a new concurrency control mechanism providing bounded locking delays, thus enabling hard real-time predictability.

2.1.2 Real-Time Object-Oriented Systems

Several prior systems have applied object-oriented techniques to real-time systems. CHAOS [12, 34], Maruti [24, 28], and ARTS [46] provide support for real-time objects at the kernel level of an operating system. MO2 is an object-oriented model with features of database systems and real-time systems [4]. MO2 supports distributed active objects with per-object read and write servers that execute client requests at the client priorities. Ishikawa *et al.* [16] describe a real-time extension of C++ called RTC++.

Each of these systems supports specification of timing information for the object methods. However, the timing information in all of these systems is expressed in seconds (i.e., time units) and hard-coded in the class definitions. This approach does not accommodate variations in hardware performance. MDARTS has a more flexible approach to timing specification that provides support for automatically benchmarking execution times and scaling timing estimates to the performance of the execution platform at runtime (see Section 5).

Furthermore, all object sharing across multiple processors in these systems is accomplished using client-server interactions. Therefore, these systems incur the overhead associated with RPC. For example, best-case CHAOS method invocation overhead on the client side alone ranges from one to five milliseconds, and ARTS overhead ranges from one millisecond for local objects to about eleven milliseconds for objects on other processors.

Stewart *et al.* describe an object-oriented approach to developing hard real-time applications in [44]. The objects, called port-based objects, share information by copying data between a global state table and local caches. Tasks run asynchronously to compute their results, which are copied out to the global table at the end of the cycle. The entire global table is locked when copies to or from it are made. The port-based object approach is appropriate for some problem domains, but the data sharing mechanism lacks flexibility and provides no support for serializing concurrent transactions. Furthermore, if applications share large amounts of data, it is expensive to copy it between the local and global state tables each cycle.

2.1.3 Client-Server Overhead

Figure 1 illustrates the types of overhead implicit in client-server architectures. We show both multiprocessor and uniprocessor examples. Figure 1 reflects the simple transaction model of MDARTS, in which each transaction is bundled into a single request and sent to the database server. The database transactions themselves are highlighted with the bold dashed lines. Each transaction is decomposed into a start-up region **S**, a critical sec-

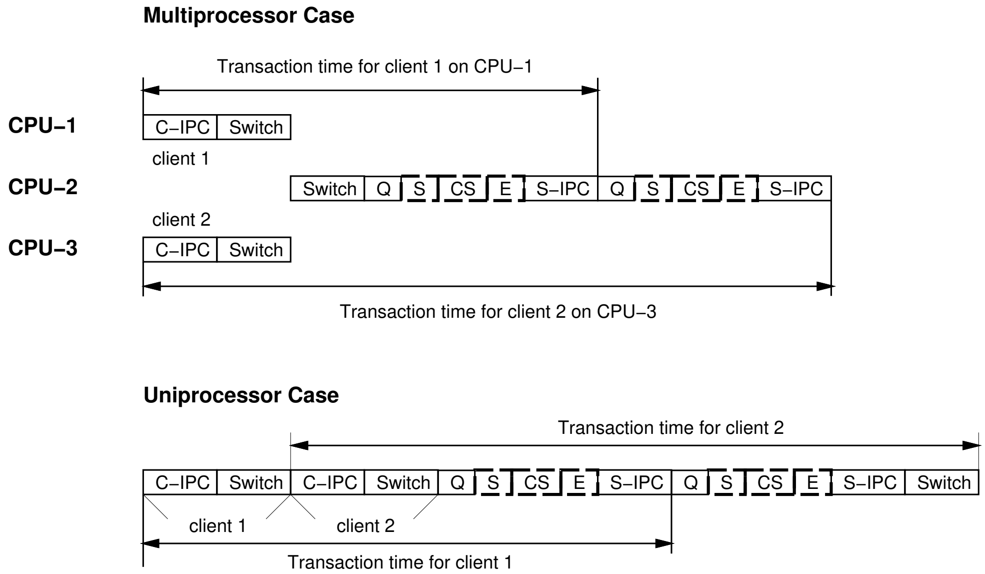

Figure 1: Real-time performance implications of the client-server architecture.

tion **CS** (in which mutual exclusion is required), and an end region **E**. The relative lengths of these regions depend on the particular transaction. The client-server overhead is labeled as follows: **C-IPC** represents client-side inter-process communication, which includes RPC stub procedure call overhead, data conversion and marshalling, copying overhead, and transmission latencies (if the RPC is not local), **Switch** represents context-switch overhead (we assume, for simplicity, that the server task requires only one context switch to service both client requests), **S-IPC** represents server-side inter-process communication, and **Q** represents time required to enqueue client requests in the server.

The relative sizes of overhead components depend on the characteristics of the target hardware and operating system. Usually, the context-switch and inter-process communication overhead is on the order of a few milliseconds, whereas the transaction execution time could be only a few microseconds. Furthermore, the client-server architecture implies a serial bottleneck in the server processes. This is not a problem on a uniprocessor, but on a multiprocessor it limits parallelism when multiple simultaneous transactions use the same object (i.e., use the same server). On a uniprocessor, more context switches are generated as the CPU switches execution from clients to servers. Note that real-time scheduling algorithms can only rearrange the order in which these operations are performed. Far better performance can be achieved if the overhead itself can be reduced or eliminated.

Figure 2 illustrates the approach taken in MDARTS. Rather than client tasks submitting requests to servers, the client tasks use MDARTS objects that point directly to shared

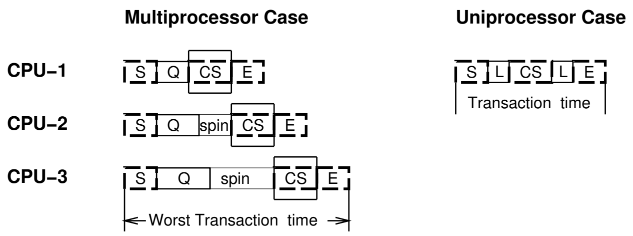

Figure 2: Transaction times if clients perform transaction execution.

memory. The client tasks can thus perform the transactions themselves, using the MDARTS object transaction methods. In the multiprocessor case, critical sections are guarded by spinlock queues (variants of spinlocks with fair scheduling policies). The locking protocols are implemented within the MDARTS transaction code, so applications need not concern themselves with these low-level issues. Note that contention to enter the spinlock queue runs in parallel with the critical section of the lock holder. Therefore, this approach makes excellent use of the parallelism available on a multiprocessor. In a uniprocessor, the critical sections are guarded by locking task preemptions. This approach is viable for short critical sections that perform memory-based operations.

Because context-switch and inter-process communication overhead is often much larger than the actual execution times of transactions, the performance gains achievable by avoiding client-server interactions can be very significant. This is especially true when individual transaction times rather than transaction throughput is considered. On our implementation platform, RPC overhead was typically three orders of magnitude greater than the basic transaction execution time. By avoiding the client-server model, we were able to achieve much better performance and predictability in our database transactions.

3 MDARTS Transactions

3.1 Transaction Model

In this section, we introduce basic definitions and notation of MDARTS transactions. MDARTS uses an object-oriented approach which models database entities as objects. In an object-oriented programming paradigm, objects consist of data and methods which are interfaces accessing the data. In MDARTS, transactions are modeled as object methods. All application tasks access databases by invoking transaction methods simply registered in each object.

Definition 1 : Each MDARTS object consists of object identifier, data, and transaction methods: $\langle OID, D, T\rangle$. Each transaction method(T) is a sequence of operations on data(D). T is defined as $\langle TID, O, Arg\rangle$, where TID is transaction method identifier, O a set of operations, and Arg a set of arguments. MDARTS objects can be accessed through only transaction methods.

In ordinary database systems, applications are permitted to explicitly control the scope and duration of transactions. An application defines a transaction by executing a Begin_Transaction() operation, performing a set of arbitrary database operations and computations, and executing a End_Transaction() operation. Clearly, this type of transaction support is extremely powerful and useful from an application's perspective. However, we believe that this level of transaction support is fundamentally incompatible with the absolute transaction-time guarantees needed by hard real-time systems.

Consider a typical scenario in which an application begins a transaction, reads some portion of the database, and then begins an extensive computation to determine derived values with which to update the database. During this extensive computation, the portions of the database that will be affected by the transaction must remain locked. Because the length of time the lock is held is not under the control of the database, there is no way the database system can provide any real-time guarantees for other transactions that need to access the locked data. The database system might be able to abort the first transaction if a higher-priority transaction came along, but this would waste resources and make it more difficult to guarantee that lower-priority transactions will meet their deadlines.

Unconstrained application-defined transactions may or may not be acceptable in the context of soft real-time systems such as those considered in [15], but to provide hard real-time guarantees, a database system must tightly control transaction execution. Therefore, MDARTS transactions are modeled as object method invocations. An application can provide parameters to the transactions, and a transaction can perform relatively complex computations.

Definition 2 : Let O be an MDARTS database object and $T_1, T_2, \ldots, T_n$ transaction methods registered in the object O. We denote $T_i \in O$, if T_i is a transaction method of O. E_{T_i} is the execution time of a transaction method T_i. We define the maximum access time of an object O, denoted as W_O : $W_O = max\{E_{T_i},\ where\ T_i \in O\}$.

The execution time of each transaction method, E_{T_i}, and the maximum access time of an object, W_O, are calibrated in advance, when an object is declared. E_{T_i} and W_O are stored into shared memory and are used to estimate the worst-case transaction execution time.

Definition 3 : Application tasks can issue MDARTS transactions by invoking transaction methods. An MDARTS transaction invocation consists of $\langle TID, Arg, TC\rangle$, where TID transaction method identifier, Arg arguments for the transaction method and D the deadline.

Application tasks can initiate transactions with a transaction method invocation specifying their arguments and deadlines. The deadline is the timing constraint that the transaction method invocation must be completed. In this paper, we will use the term "transaction" to specify transaction method or its invocation, if the context of its use is clear or the distinction between the two is not important.

3.2 Concurrency Control and Transaction Scheduling

A major source of performance unpredictability in conventional databases is the transaction delays or aborts when concurrent tasks try to acquire locks. Unpredictable locking delays in transaction processing make it impossible to estimate the worst-case execution time of each transaction. In order to provide hard real-time guarantees for transactions, it is essential to determine their worst-case execution times and to bound locking delays.

MDARTS provides a new locking protocol, *spinlock protocol*, to bound locking delays as follows.

(1) Transaction methods registered in an object are protected as critical sections by acquiring (releasing) the same lock at the beginning (end) of transactions. The object will be accessed in a mutually exclusive manner by these transactions. (We may achieve better performance if both shared and exclusive locks are supported, but, for simplicity, MDARTS uses only exclusive lock mode.)

(2) When a transaction tries to acquire a lock held by another transaction, it does not block but spins at a spinlock queue. Each transaction running on a processor is not preempted until its completion.

(3) Lock requests in a spinlock queue are processed in a FIFO order. The maximum length of spinlock queues is equal to the number (n) of processors. The locking delay will then be bounded by the time to complete the other $(n-1)$ transactions.

Under this protocol, if a transaction starts to run on a processor, no other transactions can be initiated on the same processor until the first transaction is completed. Note that each processor executes transactions sequentially, one at a time. So, a transaction may encounter lock conflicts with only those transactions running on different processors. Because a transaction holds a lock until its completion, priority inversion can occur in this protocol. However, the priority inversion can be bounded by using the FIFO policy.

Notation: The hardware platform is assumed to consist of n processors, denoted as a set $\{ p_1, \ldots, p_n \}$. $T_{i,j}$ is the i-th transaction to be executed on p_j. (We will use T_i if we don't need to identify the running processor.)

Lemma 1: The maximum length of a spinlock queue is $n-1$, where n is the number of processors.

Proof: A transaction $T_{i,j}$ doesn't have any lock conflict with the other transactions, $T_{k,j}$, to be executed on the same processor p_j, $j \neq k = 1, \ldots$, because each processor executes transactions sequentially. Each transaction $T_{i,j}$ on p_j can have lock conflict with at most $(n-1)$ transactions running on the other processors, p_k, $1 \leq k \leq n$, $k \neq j$. In the worst case, all n transactions try to access the same object O. While one of them is accessing O, the other $(n-1)$ transactions must wait in the spinlock queue for the object O. Thus, the maximum spinlock queue length is $n-1$.

Theorem 1: The worst-case execution time W_T of a transaction T is bounded to $E_T + (n-1) \cdot W_O$, where E_T is the execution time of T's method and W_O the maximum execution time of the transaction methods registered in the object O.

Proof: W_T is determined with the execution time E_T of the transaction T and its worst-case blocking time (B) as $W_T = E_T + B$. By eliminating task preemptions, the blocking time of T includes the locking delay only. As stated in Lemma 1, the maximum length of a spinlock queue is $n-1$ and the spinlock queue is processed in a FIFO order. To acquire a lock, therefore, a transaction has to wait until the completion of the other $(n-1)$ transactions. Let $T_{i,1}, \ldots, T_{i,n}$ be the transactions currently running on processor p_k, $1 \leq k \leq n$, respectively, and let $T_{i,1} = T$. Then,

$$W_T = W_{T_{i,1}} = E_{T_{i,1}} + \sum_{k=2}^{n} E_{T_{i,k}}.$$

The worst case occurs when all n transactions try to access the same object O. Because W_O is the maximum execution time of the transaction methods registered in the object O, $E_{T_{i,k}} \leq W_O$, $2 \leq k \leq n$. So, the worst-case execution time of a transaction T, $W_T = W_{T_{i,1}} \leq E_{T_{i,1}} + (n-1) \cdot W_O$.

Under the spinlock protocol, MDARTS transactions are executed in a very simple way, so MDARTS doesn't need any scheduling server that can manage multiple concurrent transactions. Instead, each task performing transactions participates in the transaction scheduling. MDARTS achieves this by embedding scheduling logic in tasks. The tasks themselves execute database transactions. The transactions are automatically performed at the priority levels of the corresponding application tasks, and no system overhead incurs to dynamically modify the task priorities.

In MDARTS, the application tasks that have already been assigned and scheduled on processors perform the transactions. If ten tasks on ten processors execute transactions on ten different database objects, all of these transactions can proceed in parallel with minimal transaction scheduling overhead. If some of these transactions attempt to use the same object, the locking protocol of that object's methods guarantees the serializability of the concurrent operations with a simple FIFO policy.

3.3 Real-Time Guarantees for Transactions

Prior real-time database systems either make deadline guarantees *a priori* with off-line static analysis of applications [38] or use dynamic transaction scheduling to try to meet deadlines at runtime [6]. Off-line static analysis has the advantage of providing early feedback if requirements cannot be met. However, off-line analysis of transactions is not always feasible, especially for complex, distributed applications. Dynamic transaction scheduling is a viable alternative for soft real-time systems. However, if deadline information is processed during transaction execution, overload conditions might cause some deadlines to be missed.

MDARTS is unique in registering real-time requirements on a per-object basis during application initialization. In other words, MDARTS performs advance Each application task includes transaction invocations inside its code. As stated in Definition 3, transaction invocations specify transaction methods, arguments, and deadlines. These transaction invocations are checked at the time of task initialization to see if their deadlines can be met. The worst-case execution time of each transaction method is calculated *a priori* as described in Theorem 1 in Section 3.2 and is stored in shared memory. For example, a task τ invokes m transaction methods $T_1, T_2, \ldots, T_m$ with deadlines $D_1, D_2, \ldots, D_m$, respectively. At the time of task initialization, MDARTS identifies the m invocations and checks their deadlines. If every $D_i > W_{T_i}, 1 \leq i \leq m$, the task can provide hard real-time guarantees for its transactions, where W_{T_i} is T_i's worst-case execution time. This approach maintains most of the flexibility advantages of dynamic deadline guarantees while making guarantees before the transactions are actually performed. Furthermore, transaction execution performance is enhanced since the overhead of checking these requirements and constructing the data access objects is incurred only once per task before the real-time processing begins.

An RTDBS for hard real-time applications must be able to provide real-time guarantees for *each* transaction. Alternatively, and just as importantly, if no guarantee can be made, the RTDBS should reflect this as well. The emphasis on *per transaction* real-time guarantees is one of the fundamental principles of MDARTS.

To implement a database service within the MDARTS framework and to provide transaction-time guarantees, it is necessary that the computational environment also be predictable. Therefore, a suitable platform for MDARTS (or indeed any hard real-time application) must provide consistent processor performance and bounded latency for bus and memory access. Furthermore, any transactions that require network communication must either be non real-time transactions (without hard real-time guarantees) or be based on networking protocols that provide end-to-end response-time guarantees, like the one described in [19]. In a shared-memory multiprocessor, it is necessary to characterize delays associated with using the shared interconnection network that provides access to the global memory. MDARTS transactions that use the multiprocessor bus must account for network access latency in their real-time guarantees. This is a factor over which MDARTS has no control, and it is highly implementation-dependent. Some interconnection buses, such as the VME bus, support DMA operations that can seize control of the bus for extended periods of time. Any computing platform with components that monopolize the bus, e.g., by performing uninterruptible DMA transfers, will severely limit the timing guarantees MDARTS can make for shared-memory transactions. However, if DMA operations are interruptible, and the bus master is configured to support a deterministic scheduling protocol (such as round-robin scheduling), it is possible to determine worst-case latencies to access the bus. Given these worst-case latencies, MDARTS can guarantee its transaction times.

3.4 Transaction Recovery

Some of the ACID (atomicity, consistency preservation, isolation, and durability) properties [10] required by traditional transactions may be too expensive or infeasible to provide in the context of a hard real-time database [26, 37]. Durability can be particularly expensive as it usually implies keeping the database on a disk, thus incurring all of the overhead and execution time uncertainties associated with disk I/O. The transaction processing speed of main memory databases will then be bounded by disk I/O speed, and hence, we can't take full advantage of the speed of main memory. Furthermore, disk I/O causes the blocking of a process, thus making it impossible to estimate transaction execution time. Despite all of these difficulties, durability is not always needed for real-time applications which require external consistency.

MDARTS has been implemented for a hard real-time database system with microsecond-order response times, but it does not implement crash recovery (that makes databases durable even in the presence of a system crash or shutdown). System crash recovery inherently requires to save all log records of committed transactions into a disk, while in a

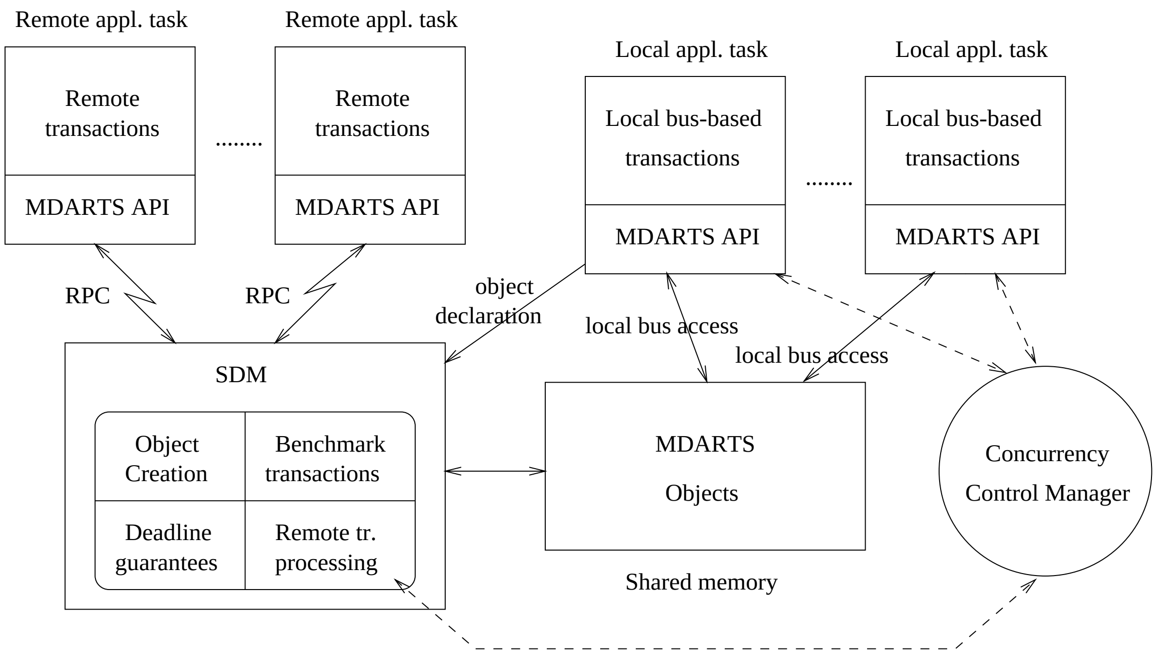

Figure 3: Overall Architecture of MDARTS.

hard real-time system each task must complete its execution before its deadline. Hence, no transaction is allowed to miss its deadline in order to avoid system failure. System crash recovery can be handled by augmenting MDARTS with such fault-tolerant approaches as replication of hardware or software.

Because all MDARTS transactions must be finished within their deadlines, each transaction is not aborted by the system or by the other transactions. Every transaction has to be committed successfully or rolled back intentionally by itself within its deadline. No other states are allowed in MDARTS. As described in Section 3.2, MDARTS transactions can access each object sequentially. Under such transaction model of MDARTS, transaction rollback can be achieved by using shadow objects. Transactions may copy of each object into its shadow object. When a transaction should stop its execution, it can be undone by overwriting the shadow object over the corresponding partially-updated object.

4 MDARTS Architecture

MDARTS consists primarily of a library of object-oriented database service classes. Figure 3 shows the overall architecture of the MDARTS system. As shown in the figure, tasks needing to share data with other tasks declare objects belonging to the MDARTS database classes. These objects are automatically registered with an MDARTS Shared

Data Manager (SDM) server 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. The MDARTS object creation process examines the constraints during application initialization and constructs data objects that satisfy the constraints. By registering application needs during initialization, MDARTS objects are able to track resource allocation at runtime and guarantee transaction times before transactions using them are actually performed. MDARTS also fully exploits the hardware capabilities of shared-memory multiprocessors by supporting both remote network-based transactions and local bus-based transactions. Local transactions access MDARTS objects in shared memory directly without inter-process communications, while remote transactions forward their requests to SDM via remote procedure calls (RPCs). In this section, we present the MDARTS architecture and and discuss implementation issues.

4.1 MDARTS Application Programming Interface

An important consideration in any database is how applications access the database. The most popular approach is to provide an interpreter for the language SQL. This language was originally created to support ad hoc queries on relational databases. Although popular, this language is awkward to use in the context of a compiled language like C or C++. Most relational database vendors provide C preprocessors that permit SQL statements to be mixed with ordinary C code. Mixing query languages and compiled languages creates the well-known "impedance mismatch" problem of database applications. The query language code that results from this awkward mixture is often inconvenient to write and inefficient to execute. Furthermore, debugging the hybrid code can be difficult.

We believe that a query language interface is totally inappropriate for high-speed hard real-time databases. The entire relational data model upon which SQL is based, with its tables, foreign keys, joins, and index searches, implies far too much overhead for this domain. In MDARTS, each object supplies the context and identity of its particular data. Therefore, it is unnecessary to perform index searches on keys to locate the data needed for each transaction. Instead, the methods of an object can follow direct memory pointers to perform transactions.

Figure 4 illustrates the MDARTS C++ application programming interface (API). Two MDARTS classes corresponding to the same database object are shown in Figure 4: Array<T> and ReadOnlyArray<T>. These are C++ template classes, where <T> indicates an arbitrary class or structure 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

```
/**********************************************************
*  Declaration of MDARTS object in sensor task that will be updating it:
*/
Array<Point>  position_sensors("position_sensors",
    "exclusive_update; size = 6; write(element) <= 50usec",  CREATE);

/*  Sensor task updates the data:
*/
position_sensors[5]  =  Point(1.2, 0.866, 3.4);
```

```
/**********************************************************
*  Corresponding declaration of MDARTS object in control task:
*/
ReadOnlyArray<Point>  position_sensors("position_sensors",
    "read(element) <= 80usec");

/*  Control task reads the data:
*/
int  i  =  position_sensors("size") − 1;

Point  end_effector_position  =  position_sensors[i];
```

Figure 4: MDARTS C++ application programming interface.

manage arrays of Point objects in Figure 4 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.

MDARTS object declarations include two string parameters that are passed to the object constructors. The first parameter is a unique identifier for that object in the database. The second parameter is a "contract" composed of 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 timing constraint in this case refers to a bound on the magnitude of the sum of the two MDARTS transaction time components (local execution time and blocking time).

The "exclusive_update" constraint specified by the sensor task in Figure 4 causes MDARTS to reject subsequent attempts to construct objects that could modify the data. This constraint allows the Array<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 [49], 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 "exclu-

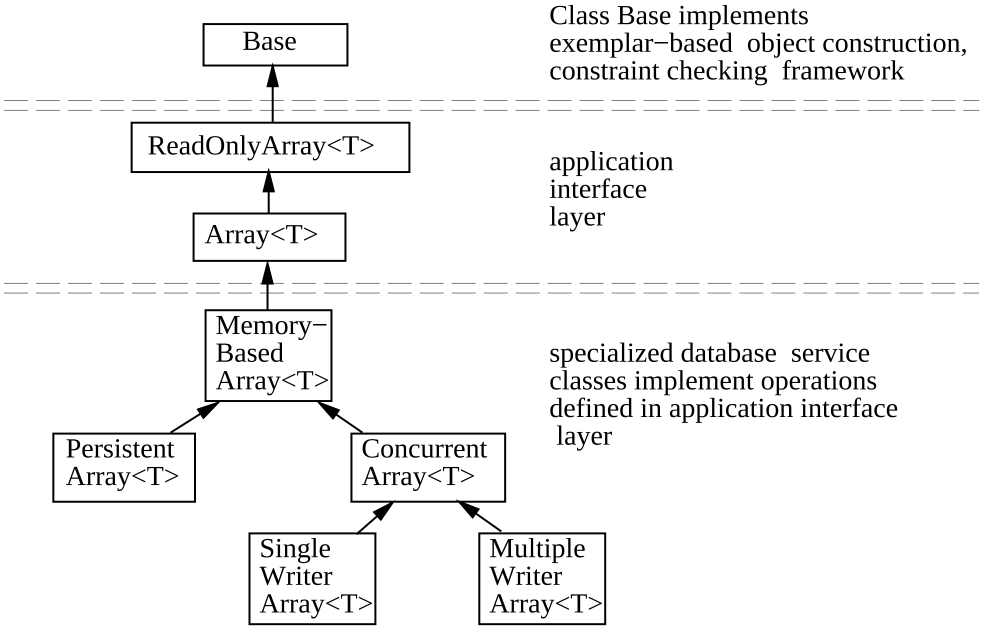

Figure 5: MDARTS database class hierarchy.

sive_update" are checked only during initialization of the data objects. Subsequent database access using the objects is not burdened with the overhead of checking access permissions. In Figure 4, the control task declaration specifies its object as a ReadOnlyArray<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 object, an error is reported at compile time.

We implemented the ReadOnly semantic restriction as a separate class so that access violations can be detected by the compiler and during the application initialization process. With this technique, it is not necessary to check access permissions during real-time transaction processing. Note that the ReadOnly class used by the application is an interface class that delegates the actual transaction performance to another class (the database service class). The database service class for a given object is always the same across different application tasks, but different (compatible) interface classes can be used to access the same service object.

4.2 MDARTS Class Hierarchy

Figure 5 illustrates the structure of the MDARTS class hierarchy. The service classes in the hierarchy, such as "Multiple Writer Array<T>," all provide the same data access operations, but each is specialized to support different constraints. 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 that supports only one writer might be declared as "DbPersistentExclusiveUpdateInteger my_object." This name might be deemed too long and be converted to something cryptic like "DbPEUInt my_object." In either 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. Furthermore, attributes that correspond to continuous variables, such as transaction times, cannot be encoded into class names.

In MDARTS, semantic attributes like "persistent" and "exclusive_update" are passed as strings to the library's object construction methods instead of being encoded into the database class names. MDARTS thus dramatically reduces the number of different database classes to which applications are exposed and thereby reduces dependencies between applications and the internal organization of the database library. Applications are only exposed to classes in the database hierarchy that correspond to different data interfaces (e.g., a floating point array vs. a linked list of strings).

Applications use abstract interfaces in MDARTS by creating objects from the classes in the application interface layers. The constructors for these interface classes forward the constraints specified by the application to the constructors of the specialized database classes derived from the interface classes. Once an acceptable specialized database service object is constructed, the interface object used by the application forwards transaction requests to that service object. This forwarding of transaction methods is a form of delegation. With C++ in-line functions and an optimizing compiler, very little runtime overhead is added through transaction forwarding.

The database service classes in the MDARTS library function like populations of individual contractors. Each MDARTS class is free to specialize services, such as concurrency control, to match particular application needs. Application-specified 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 transaction times 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 [49, 50]. 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

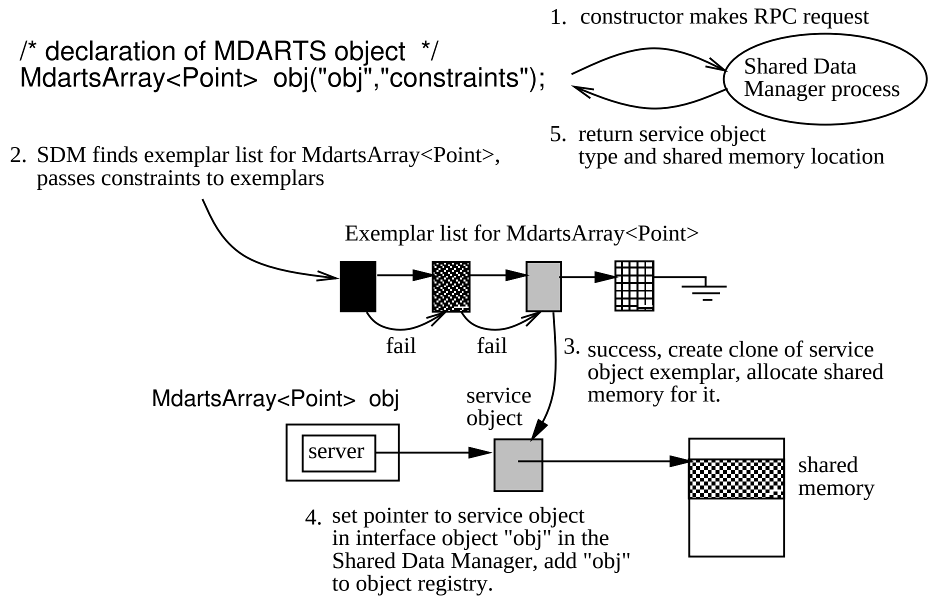

Figure 6: MDARTS object construction using exemplars.

restricted concurrency semantics as well as classes that support more general semantics. Each database class guarantees transaction times according to its own implementation.

4.3 Exemplar-Based Object Construction

Figure 6 illustrates the object creation sequence in an MDARTS shared data manager. The application declares an object, and the constructor for that object forwards the type information of its class (the application interface class) and the object's name and contract string to a Shared Data Manager server. The SDM uses exemplar-based object construction to select and instantiate a service object that meets the application needs. In exemplar-based object construction, each class has associated with it a unique instance of that class called the "exemplar." The exemplar is used to determine if a given class can meet all of the constraints specified by the application. During object construction, the MDARTS SDM submits the application contract to the list of exemplars attached to the interface class used by the application. An exemplar from this list is chosen in this contract checking stage, and the SDM clones the chosen exemplar to create a new object of that type. The SDM also allocates the shared memory needed for that object and returns the type of the object and the shared memory address to the application task in its RPC reply. After the RPC returns, the application task completes the construction of its local instance of the object. One of the key advantages of the exemplar-based approach is that it facilitates the integration of new classes into the database library. The exemplars of the new classes are simply added

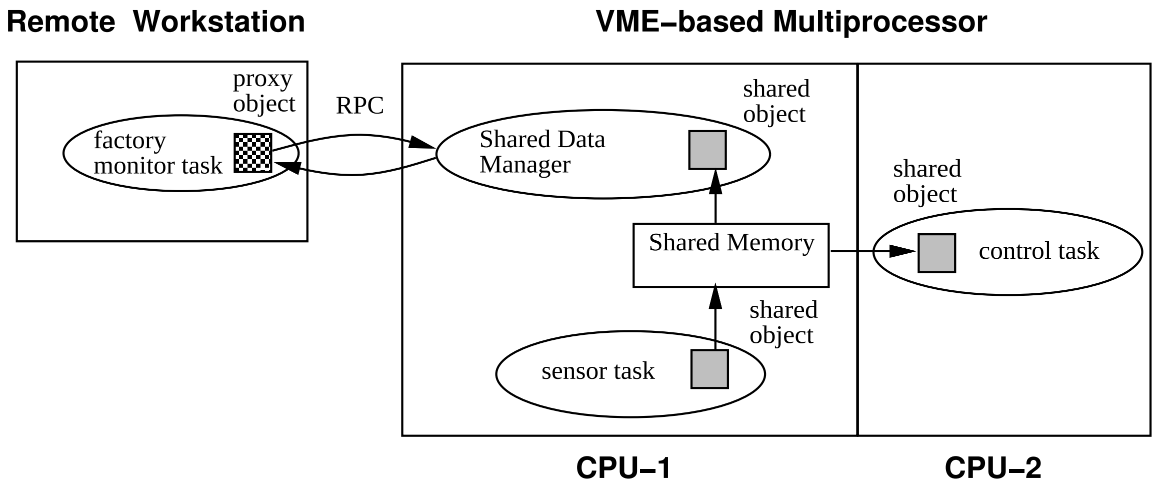

Figure 7: Access to shared memory data.

to their exemplar list, and the library functions that construct MDARTS objects require no modification or recompilation.

4.4 Shared-Memory Objects

Hard real-time systems often share data among different tasks by directly using the physical memory addresses of shared data objects. This approach has a danger that some of the tasks will inadvertently misuse and possibly corrupt the common data areas. To resolve this problem, MDARTS encapsulates access to the common shared memory using object-oriented techniques. Since all manipulation of the data is performed by the object methods, application code never uses the raw memory addresses. The object methods can thus ensure that the shared data is accessed consistently by all tasks. Using shared objects rather than raw shared memory is appealing, but many complexities arise in its implementation. MDARTS provides applications with the convenience of shared objects without exposing them to the complexity of their implementation.

Figure 7 shows an MDARTS Shared Data Manager and three application tasks sharing a common object on a shared-memory multiprocessor. The shaded boxes in each task on the multiprocessor represent local MDARTS objects that contain internal pointers to a common data structure in shared memory. The arrows in the figure represent data flow to and from the shared memory or across the network. In this example, an exclusive update constraint has been declared by the sensor task, so only it is allowed to write updates into the shared memory (hence the direction of its arrow vs. those of the other tasks in Figure 7). One of the application tasks (the factory monitor task) is running on a remote computer, so it uses a proxy MDARTS object that uses remote procedure calls to forward transactions to the Shared Data Manager. The Shared Data Manager uses its instance of the MDARTS object to perform the actual transaction for the remote task. The object instances used by the SDM and the two application tasks on the multiprocessor point to the same shared-memory region, so data consistency is guaranteed across the tasks.

Note that once the MDARTS object is constructed on the multiprocessor, transactions performed by local tasks require no inter-process communication. In this case, the MDARTS transactions are ordinary C++ function calls performed by the application tasks. This avoidance of inter-process communication is extremely important, and it is the primary reason MDARTS can achieve such high performance on multiprocessors.

One might think that it would be trivial to place C++ objects in shared memory and then use pointers to access them from different tasks. However, because of address space differences across tasks on different processors, shared memory cannot be used as easily as ordinary process memory for instantiating C++ objects. The fundamental reason for this is that pointers embedded in objects cannot be shared easily across different processes. In C++, objects usually contain pointers to functions (most often in the form of a pointer to a virtual function table). In general, these functions will be loaded at different addresses in each process, so no single function pointer will be valid for all processes. Jordan [18] 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. One portable solution to this problem is to establish a policy that forbids objects from using virtual functions or otherwise containing function pointers. However, without virtual functions, C++ loses one of its most powerful features.

Therefore, MDARTS places only the shared data parts of objects in shared memory. The rest of the MDARTS objects, including the virtual function pointers, are instantiated as ordinary C++ objects in local process memory. This permits the portable use of C++ object-oriented features together with shared memory. Each local MDARTS object corresponding to a particular database object has an internal pointer to the same region of shared memory. A given application may consist of many separate tasks, each of which shares the same data objects through their local MDARTS object instances. Concurrent access to the shared memory region is managed by the implementations of the MDARTS transaction methods of each object.

5 Benchmarking Execution Times

We have discussed the basic mechanisms in MDARTS for expressing and checking timing constraints, but we have yet to consider how a database class can know its own performance so that it will make accurate transaction-time guarantees. A similar problem is addressed in the RTC++ language [16]. In RTC++, the designer of a real-time class specifies the worst-case execution time bound of each method in the class definition. An example given in [16] is: `int m33(float f) bound(0t30m);`. This declaration specifies a 30 millisecond bound for the method `m33`. This method of determining performance has three serious limitations. First, it requires the class developer to determine the execution time bound "by hand" using some unspecified method. Second, it ignores the possibility of heterogeneous computing platforms and hardware evolution. The execution times of methods will vary over hardware platforms. The specification of worst-case execution times is useless in different hardware platforms. Third, the timing specification method in RTC++ does not account for blocking delays associated with synchronization. Analysis of "nonpreemptive objects" that can cause blocking must be performed manually.

In this research, we prefer to empirically measure the execution times of MDARTS object methods. Given a clock with sufficient resolution and transaction methods that exhibit predictable performance in the absence of contention, an empirical approach is sufficient to characterize execution times. We believe that most database transactions will consist of simple code sequences that, apart from concurrency control delays, have highly predictable performance. By benchmarking execution times, we can automatically factor in the CPU speed and other attributes of the execution platform. Benchmarking addresses the first two drawbacks of the RTC++ approach. Combining the benchmarking results with runtime lock information addresses the third.

5.1 Benchmark Design

Transaction-time guarantees in MDARTS are derived from benchmarking of method execution times, runtime estimation of worst-case locking delays, and estimation of worst-case bus access times. An MDARTS database service class includes a virtual function called `calibrate()` that performs a set of timing experiments on its methods. These timing experiments can be performed in a separate calibration run. The results of the calibration can be output in a form that is included in the class source code, which is then recompiled.

With calibration runs, MDARTS could be vulnerable to the RTC++ problem with respect to a heterogeneous computing platform (e.g., a mixture of 68030 and 68040 processors). There are two approaches to this problem in MDARTS. The first is to perform multiple calibration runs, one on each platform. The second approach is to scale execution times in units of the execution time of a standard function. This permits automatic scaling of transaction times to the execution speed of the CPU. For example, suppose the standard function is timed at 40 microseconds on a 68030. If a calibration run on that CPU measures method M at 20 microseconds, it could output the execution time as 0.5 benchmark units (which we abbreviate *bms*). Suppose this code is then run on a 68040 on which the standard function requires only 10 microseconds (four times as fast). Then the MDARTS library on the 68040 would infer that method M will require 5 microseconds.

MDARTS performs two experiments for each benchmark in a calibrate() function. The first experiment times the overall execution of the method in the absence of concurrency control delays. The second experiment measures the maximum critical section time $CSTime$ and the number of critical sections entered by the method. The critical section information is collected by the lock objects used to control access to the critical sections. When critical section timing is enabled, the getLock() method of each lock reads a hardware timer, and the releaseLock() method reads it again to measure the length of the critical section. GetLock() also increments the critical section counter.

The local execution transaction time for a database transaction is the time required for its execution plus the time spent busy waiting in the spinlock. If the execution time is $EXtime$, the number of critical sections is NCS, and the wait time to acquire a lock (to enter a critical section) is bounded by D, then the overall transaction time is bounded by $EXtime + (NCS \times D)$. Included in $EXtime$ is a bus access factor that accounts for worst-case latencies to perform whatever bus operations are required by the transaction. The lock delay bound D depends on the locking protocol. In our spinlock implementation, D is bounded by $Q + (MaxCSTime \times N)$, where Q is a fixed queuing overhead, $MaxCSTime$ is the maximum critical section time and $N - 1(???)$ is the number of CPUs running tasks that share the object. Figure 2 in our discussion of client-server overhead illustrates the bound on D provided by spinlock queues.

The examples of timing constraints presented thus far, such as "read<=30msec", have implied a rather coarse view of object transaction methods. Suppose the transaction time of an MDARTS object depends upon which member is accessed. If an application can only specify a single timing constraint for all read or write transactions for an object, the object must make the pessimistic assumption that the most time-consuming transaction will be performed. This may cause an unwarranted rejection of a timing constraint that could actually be met if it were known which transaction would be performed. Thus, MDARTS permits each transaction method to have as many parameter-specific timing records as the class implementer deems worthwhile. With this mechanism, the MDARTS bechmarking

```
char * MDclass::Timep[] = {
"read(delay);20usecs;1;0usecs;0",
"read(name);40usecs;10;0usecs;0",
"read(sum);0.5bms;1x;0.1bms;1",
"write(start_motors);10msecs + 3bms;2;50usecs;2",
"write(increment);0.4bms;2x;0.1bms;1",
0 };
```

Figure 8: Example of MDARTS benchmark data.

approach allows applications to specify timing constraints of very fine granularity, such as "read(update)<1msec; read(sum)<50usec; write(increment)<1msec".

5.2 Benchmark Implementation

Benchmarks in MDARTS are stored as sets of records that contain the name of the transaction, the execution time, the bus operations and size scale factor, the maximum critical section time, and the number of critical sections entered by that benchmark. When output by a calibration run, this information is encoded into an array of character strings associated with that database class. Figure 8 illustrates a set of benchmarks for a hypothetical MDARTS class. Each benchmark record is composed of five fields delimited by semicolons. At application initialization time, the array of strings is processed and converted into an internal format that efficiently supports retrieval of transaction times by name. The first field is the name of the transaction. By convention, this name is prefixed by either "read" or "write," and it includes parameters such as the name of the data field being accessed. The second field is the execution time for the transaction in the absence of concurrency control delays.

The third field contains the number of bus operations and an optional scale modifier for the transaction. The scale modifier indicates a scale factor for execution time and bus operations. A constant scale modifier is used to adjust the time bound to some constant factor of the time measured during calibration. This technique can be used to make the guaranteed times more conservative. Modifiers with the letter "x" in them correspond to size scaling. For example, a "sum" transaction will access all of the elements in an array, thus its number of bus operations will be proportionate to the size of the array. The fourth field in the benchmark record is the execution time of the longest critical section entered by the transaction. The last field is the number of critical sections entered by that transaction.

Figure 9 shows the implementation of a calibration function corresponding to Figure 8. Notice that some of the benchmarks are in terms of microseconds or milliseconds while others are in terms of "bms", which is the execution time required to execute a standard

```
static void MDclass::calibrate() {
    int j;
    char buf[80];
    CALIBRATE_START(MDclass)
    RUN("read(delay)",j = getIValue(delay_f,"",0),"usecs")
    RUN("read(name)",getSValue(name_f,"",0,0,buf,80),"usecs")
    RUN("read(sum)",j = getIValue(sum_f,"",0),"bms")
    DECLARE("write(start_motors);10msecs + 3bms;2;50usecs;2")
    RUN("write(increment)",setValue(increment_f,"",0,j),"bms")
    CALIBRATE_END
}
```

Figure 9: Example of MDARTS calibrate function.

benchmark function. An application will generally express its transaction time requirements in ordinary time units such as microseconds or milliseconds, so MDARTS automatically converts benchmark units to time units when the calibration data is processed at application initialization time. The "write(start_motors)" benchmark string is actually generated by the DECLARE macro. DECLARE allows the class programmer to generate a hard-coded benchmark during calibration without actually executing the operation. This capability can be useful in some contexts, especially if there is some side effect of executing the operation that would be undesirable during calibration.

```
#ifdef CALIBRATE
#define CALIBRATE_START(c) printf("char * c::Timep[] = {\n");
#define CALIBRATE_END printf("0 };\n");
#define DECLARE(dec) printf("\"%s\",\n",dec)
#define RUN(name,call,units) \
    { int tot; double cs; \
    Time unit_time = TimeList::ConvertToTimeUnits(1.0,units); \
    Lock::TurnOffMonitoring(); GET_COUNTER(tot); call; DELTA_TO_TIME(tot); \
    resetBenchmark(); Lock::PrepareMonitoring(); call; Lock::TurnOffMonitoring(); \
    printf("%c%s;%6g%s;%d%s;%6g%s;%d%c,\n",'"',name,(double)tot/unit_time,units,\
    busOperations(),scaleFactor(),\
    (double) Lock::CsectionTime()/unit_time,units,Lock::CsectionCount(),'"'); }
#else        // non-calibration version of the code
// declare null versions of these macros
#define RUN(name,call,units)
#define CALIBRATE_START(c)
#define CALIBRATE_END
#define DECLARE(dec)
#endif
```

Figure 10: Macros used in MDARTS calibration functions.

The macros used in Figure 9 are defined in Figure 10. These macros are the ones used to generate output during a calibration run for an MDARTS class. The RUN macro is complex enough to merit detailed explanation. Run takes as parameters the name of the benchmark, the method call corresponding to it, and the units in which to specify the timing results. It first determines the execution time of the units by calling an MDARTS function called ConvertToTimeUnits(). This function returns an unsigned 32-bit integer corresponding to the time values passed to it.

RUN next turns off critical section timing in the MDARTS Lock classes. RUN then executes and times call. Now that the overall timing of the call is completed, RUN resets the bus operation counter and the scale factor, instructs the Lock class to prepare for critical section timing, and executes call once more. The transaction methods in call count bus operations and set the scale factor. The MDARTS Lock classes used in call automatically determine the number of critical sections entered and the worst-case critical section time. Finally, RUN prints the timing results in a form that is ready for inclusion into the source code of the class that is being calibrated. Note that the time values output by RUN are scaled to correspond to the units specified in the calibrate() method. To do execution timing, RUN uses two other macros, GET_COUNTER and DELTA_TO_TIME. On the 68030 boards of our multiprocessor, these macros use a hardware timer with resolution of 6.25 microseconds.

The MDARTS approach to determining object method performance has significant advantages over the approach of RTC++. Our method requires less effort on the part of the database class developer, and it permits the specification of very fine-grained timing constraints. For example, in RTC++ only one time bound may be associated with a method. In MDARTS, multiple benchmarks can be performed for the same method using different parameters. Applications can then specify detailed timing constraints such as: "read(update)<1msec; read(sum)<50usec; write(increment)<1msec". Furthermore, by counting bus operations invoked by methods and including scale factors for execution times that depend on the data structure size, MDARTS provides much better estimates of performance than can be expressed in RTC++.

In addition to calibrate(), MDARTS supplies a function called Time QueryTiming(char *transaction_name), which returns the guaranteed execution time of the specified transaction. This function first retrieves the benchmarked timing values for that transaction. Next, it queries the object's lock for D, the worst-case spin delay for that lock. Finally, QueryTiming() computes and returns a worst-case execution time for that transaction based on the nominal execution time, bus latency, and bounded synchronization delay. QueryTiming() is used by MDARTS during constraint checking to determine if transaction-time constraints can be satisfied. Applications can also call QueryTiming() directly after the object is created.

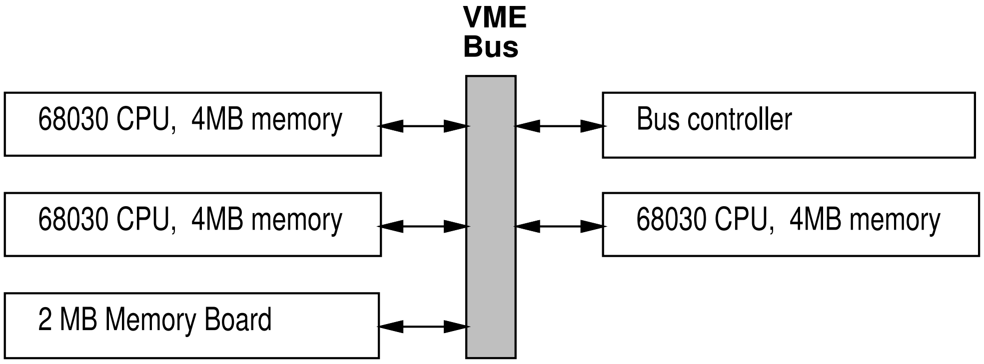

Figure 11: Platform used for MDARTS evaluation.

This permits applications to perform more sophisticated timing analysis and to specify their minimum timing requirements.

6 MDARTS Performance Evaluation

To evaluate the performance of our MDARTS prototype, we conducted a series of experiments designed to create worst-case contention and transaction loads. Figure 11 depicts the hardware platform we used for our experiments. The experimental platform was a standard VME-based multiprocessor with a bus controller, three Motorola 68030 processor boards, and one standalone memory board with two megabytes of RAM. Each of the processor boards was running the VxWorks real-time operating system. VxWorks provides a UNIX-like environment with networking support and an efficient kernel that performs fixed-priority preemptive scheduling of user tasks. Each 68030 board has its own Ethernet interface. The VME chips on the 68030 boards also provided support for inter-processor interrupts across the VME bus. Note that this is a commercially available multiprocessor system. No specialized hardware was purchased or developed to support our MDARTS implementation.

The VME bus on our platform is clocked at 16 Megahertz, so there are 16 bus cycles per microsecond. The maximum throughput of this bus is 40 megabytes/second, although typical throughput is more like 20 megabytes/second. Therefore, five 32-bit bus operations can typically be performed in a microsecond. Two of the 68030 processor boards have a 20 MHz clock rate. The other 68030 has a 25 MHz clock rate. Clearly, any VME bus access is limited by the bus bandwidth, so one cannot gauge system performance solely by the processor speeds.

Since MDARTS transactions on multiprocessors use global shared memory, the latency to access that memory is an important component in any transaction-time guarantees. Therefore, it is necessary to bound this latency as tightly as possible. To this end, we took two steps. First, we configured the VME bus controller to grant bus requests in round-robin order. That way, given n processors, a given processor board would at most wait for $n-1$ other VME operations before it was able to use the bus. Second, we allocated memory for the shared data structures of MDARTS objects on the auxiliary memory board. This eliminated the contention for the local bus of a remote processor that occurs when remote processor memory was accessed.

6.1 Experiment Design

Clearly, the worst-case transaction load condition is when tasks on all of the CPUs try to perform conflicting transactions on the same object simultaneously. To create this maximum load condition, it is necessary to synchronize the execution of tasks on different CPUs and to perform multiple transactions in a tight loop. We decided to use interprocessor "mailbox" interrupts available on the 68030 boards to synchronize execution of competing transactions on multiple CPUs.

Conventional database benchmarking measures average transaction throughput in transactions per second. Most prior real-time database research compares different transaction scheduling algorithms in terms of the fraction of transactions that meet their deadlines as the load increases. The former metric is suitable for non real-time applications, and the latter is suitable for soft real-time applications, but neither is appropriate for hard real-time applications. In a hard real-time environment, every transaction must meet its deadline. Therefore, we decided to measure the elapsed time of every MDARTS transaction in our experiments. Each of our processor boards has a hardware timer chip with a resolution of 6.25 microseconds. With this timer, we were able to measure individual transaction times.

Since each MDARTS class encapsulates its concurrency control and transaction implementations, each must be individually tested. Therefore, our strategy for testing the MDARTS objects was to simultaneously perform identical transactions on the same object from tasks on different CPUs. We needed to develop techniques for constructing the objects, synchronizing their transactions, and collecting the resulting timing information. We also wanted to have a flexible means of conducting multiple timing tests without compiling separate programs for each test.

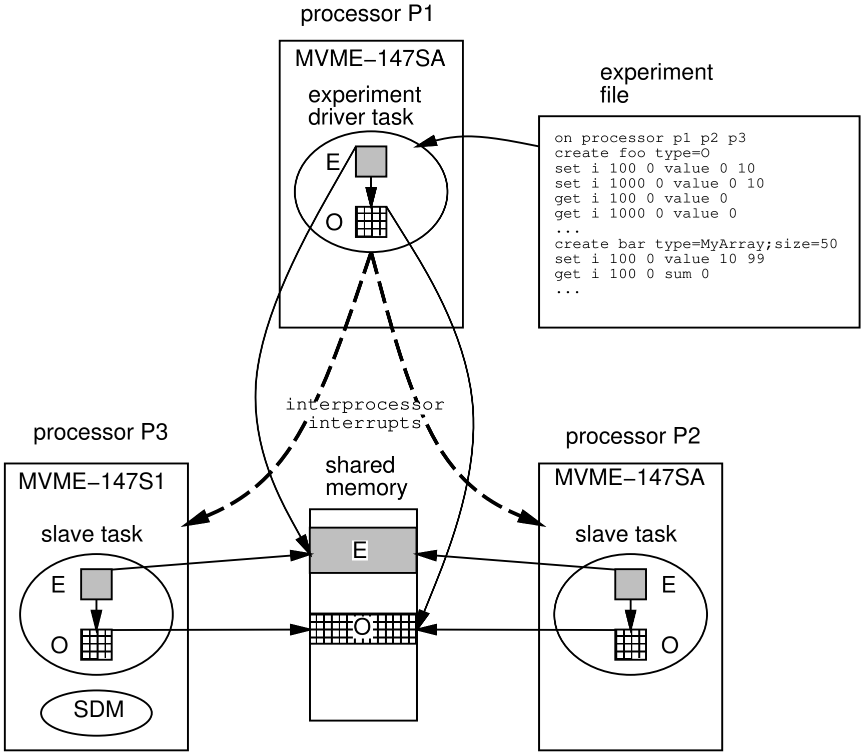

Figure 12: MDARTS experiments.

6.2 Experiment Implementation

Figure 12 illustrates the approach we used to perform our experiments. We first created a special-purpose MDARTS class called Experiment (objects of class Experiment are labeled E in Figure 12). Each Experiment object contains a pointer to another MDARTS object (labeled O). The shared-memory part of an Experiment object contains fields that specify which experiment to run and which parameters to use to construct new O objects or perform transactions using the O objects. One of the CPUs ran a task called the experiment driver task. This task contained an interpreter for a very simple experiment specification language. With this language, we defined a set of experiments in an input file. As the experiment driver task read this input file, it used its Experiment object to store experiment parameters in the database. The other two CPUs ran slave tasks that also shared the Experiment object E. These slave tasks waited for a signal from the experiment driver task to perform their transactions.

Once the parameters for a given experiment were in place, the experiment driver task signalled the slave tasks on the other CPUs using a mailbox interrupt across the VME bus. When the signal was given, the three tasks performed their experiments in parallel and

```
void    Experiment::setValue(int itag, const char *tag, int index, int val) {
    // ... code deleted
    // perform getIValue experiments (got here from switch on command type)
    for (i = 0; i < repeat_count; i++) {
        GET_COUNTER(time);
        int v = obj_ptr→getIValue(the_itag,the_tag,the_index); // experiment
        DELTA_TO_MICROSECONDS(time);
        // save results of this experiment
        if (time > the_worst)
            the_worst = time;
        if (time < the_best)
            the_best = time;
        the_sum += time;
        }
    // ... code deleted
}
```

Figure 13: Experiment class method for getIValue() transactions.

recorded their transaction response times (storing them in the Experiment object).

Note that these experiments measured wall clock time rather than true MDARTS transaction times (pure execution time plus blocking time, if any). Unfortunately, some transactions were occasionally preempted by other tasks or by the operating system scheduler. These preemptions inflated the apparent execution time of those transactions. It is crucial to understand that these outlier measurements are not worst-case MDARTS transaction times. In addition to the transaction execution time, they include execution times of higher-priority tasks on the local CPU. The scheduler interrupts and preemptions were rare, so they did not greatly affect the average transaction times. Nevertheless, their presence prevented us from empirically measuring the worst-case MDARTS transaction times.

The code segment in the Experiment object that performs getIValue transactions is shown in Figure 13. GetIValue is the MDARTS transaction method that retrieves integer values from an MDARTS object. We multiplex all integer reads through getIValue to simplify the application programming interface and to reduce the number of RPC functions exported by the Shared Data Manager. Once all repeat_count experiments were run by each CPU, the overall worst, best, and largest sum of execution times for all CPUs was updated in the Experiment object. The experiment driver task waited until it and the slave tasks finished the experiment (this was detected by watching a counter in the Experiment object), and then it printed the overall timing results. The object used to perform transactions (labeled O in Figure 12) could be changed many times within the input file processed by the experiment driver task, so a large number of experiments covering as many different MDARTS classes as desired could be performed without recompiling the test programs.

7 Experimental Results

7.1 The MdartsInt class

Our first experiments used a very simple MDARTS class called MdartsInt. MdartsInt contains a single shared integer variable, and its getIValue() and setValue(...,int val) methods get and set the shared integer with no locking. Since the get and set methods of MdartsInt are so trivial, the timing experiments on this class essentially reflect the execution time overhead associated with performing transactions using the MDARTS methods. Table 1 summarizes the results of our initial experiments with MdartsInt.

CPUs	operation	best	average	worst	outliers
1	get	12	13.6	18	0 > 100
	set	12	13.9	18	0 > 100
2	get	12	14.6	700	2 > 100
	set	12	15.7	706	2 > 100
3	get	12	14.8	1018	3 > 100
	set	12	15.1	850	2 > 100

Table 1: Read and Write Wall Clock Times (in Microseconds) for MdartsInt Object.

Each row of Table 1 corresponds to 1,000 transactions per CPU. The first column indicates how many CPUs were used in the experiment (the first two experiments used only one CPU, the next two used two CPUs, etc.). Thus, the first two experiments performed 1,000 transactions, the next two performed 2,000, and the last two performed 3,000.

With a single CPU, all of the transactions completed within 18 microseconds (three 6.25 microsecond ticks of the timer). However, when multiple CPUs were added, some of the transactions appeared to take several hundred microseconds. The get and set methods of MdartsInt contain no loops or branches. They simply issue VME read or write operations, which on our platform usually required less than a microsecond. The bus controller was configured to grant requests using round-robin scheduling, so bus access was not a problem. It is impossible for the MdartsInt transactions themselves to vary this much in execution time (VxWorks does not use virtual memory, so we are not seeing page fault effects). Therefore, we conclude that the apparent transaction time variations are actually due to task preemption. Since we were not running any other application tasks, the preempting tasks were operating system tasks, probably related to ethernet activity. We measured wall clock time (using a hardware timer) from the start to the end of each transaction. If the

VxWorks scheduler suspended the experiment task to run some other task, the timer kept running until the experiment task resumed execution and read the timer. The last column in Table 1 counts the number of transactions measured at more than 100 microseconds. Only two of the 2,000 two-CPU transactions and two or three of the 3,000 three-CPU transactions were outliers by this definition. Most likely, one transaction per CPU was preempted while the timer was running.

These outliers do not completely characterize the execution time variance, but it does help distinguish rare events such as task preemption from true transaction execution time variance. The average-case measurement in Table 1 are inflated somewhat by the inclusion of outliers caused by task preemption. Therefore the true best-case times and average-case times are even closer than our data indicate. If best-case times are close to average-case times, this implies that worst-case times are also close to the average-case times. Furthermore, we deliberately created worst-case contention conditions in our experiments thus the average-case transaction times are fairly close to the true worst-case times. This is a very desirable property for hard real-time systems.

If locks are used to enforce serializability, the problem of preemption or interruption becomes more significant because remote preemptions of lock holders can affect the execution times of local transactions. A compromised approach we have taken in our implementation of locks in MDARTS is to disable task switching during critical sections. This prevents long delays due to preemption of a lock-holding transaction but still allows interrupts to service critical system functions.

The MdartsInt class is so simple that it is easy to analyze its execution time. If an MdartsInt transaction is allowed to run uninterrupted, the most significant execution time variance is due to VME bus latency. The other sources of variance, CPU caching and local bus access latency, are much smaller than the VME bus latency. In our system, the VME bus controller is configured to do round-robin bus scheduling, so in the worst case an MdartsInt transaction will have to wait for the two other 68030 CPUs to finish using the VME bus. The VME bus access delay is slightly unpredictable, but under almost all cases it will be no more than a microsecond.

7.2 An MDARTS Array

Because the MdartsInt class does not use any locking for concurrency control, we performed another set of experiments on an array class that uses locks. As with the MdartsInt experiments, each experiment performed 1,000 transactions per CPU. In addition to range-checked get and set transactions on individual elements, these array classes support "size", "sum", and "increment" transactions. In all of our experiments, the arrays contained 10 integers. Although locking is not strictly necessary for read and write of integers across the VME bus, we locked the "get" and "set" operations. Arrays of more complex structures would need to do locking on individual element operations, and doing so in our integer arrays helps characterize the locking overhead in our various lock implementations. The "size" transaction returns the number of elements in the array. No locking is used for "size" transactions. The "sum" transaction locks the array, sums the values in all array elements, and returns the sum. The "increment" transaction adds an application-specified value to each element in the array. We used exclusive locks even on read-only transactions. Our objective here was to experiment with different critical section lengths rather than trying to develop realistic transaction semantics. Figure 14 shows the getIValue() method for the MdartsArray class.

Table 2 summarizes our experiments on the MdartsArray class. This class uses a spinlock queue that is designed to minimize bus traffic generated by tasks in the queue. This spinlock also disables task switching when tasks request the lock. This prevents unbounded priority inversion due to remote lock-holding tasks being preempted and blocking tasks on other CPUs. Note that the "size" transaction does not acquire a lock since there is no critical section for this read-only transaction.

The best case performance in the three CPU case were substantially better than in the one or two CPU cases. This is because the third CPU board was a 25MHZ machine, whereas the other two CPUs were 20 MHz machines. This explains the best-case performance jump for three CPUs. It really just corresponds to the performance of CPU 3 when it ran a transaction with no contention. We synchronized our experiments to maximize contention, but it is not surprising that some of the 1,000 experiments run on CPU 3 encountered no contention from the other CPUs.

Although task switching is disabled in the critical sections, interrupts are not disabled. Therefore, some jitter appears in the transaction time measurements due to tasks being interrupted while the timer is running. Careful examination of the data shows two levels of granularity: the small jitter granularity was about 70 microseconds; the large jitter granularity was about 800 microseconds. The 70 microsecond jitter was due to interrupt processing for the operating system task scheduler. The system task scheduler is triggered by a clock interrupt every 16 milliseconds. If one examines the top two rows in Table 2, one can see that on average the transactions required about 75 microseconds, but five transactions measured more than 100 microseconds. Since we are executed 1,000 transactions in this experiments, the total wall clock execution time was about 75 milliseconds. In 80 milliseconds, five scheduler interrupts will occur, so we conclude that scheduler interrupts

```
int     MdartsArray::getIValue(int itag, const char * tag, int index)
{
   int tid;
   if (itag <= 0) {    // an unspecified field number, examine the tag string.
      if (itag < 0)   // if a query for the field number of that tag.
         return getFieldNum(tag);
      itag = getFieldNum(tag);        // convert tag to field number
      }
   switch (itag) {        // note:  shared points to the shared memory region.
      case value_f:        // get the value of array element[index]
      if (index >= 0 && index < shared→lsize) {
         tid = shared→lock.getLock();
         int retval = theArray[index];  // theArray points into *shared
         shared→lock.releaseLock(tid);
         return retval;
         }
      else {
         cerr << "out of range read: " << index << endl;
         return −1;
         }
      case size_f:        // get size of array
         return shared→lsize;
      case sum_f:        // get sum of array elements
         int thesum = 0;
         int the_size = shared→lsize;
         tid = shared→lock.getLock();
            for (int i = 0; i < the_size; i++)
               thesum += theArray[i];
         shared→lock.releaseLock(tid);
         return thesum;
      default:
         cerr << "invalid field for getIValue: " << itag<< ','<< tag << endl;
      }
   return −1;        // default clause could throw an exception and never reach here
}
```

Figure 14: GetIValue() method for MdartsArray.

CPUs	operation	best	average	worst	outliers
1	get	75	76	193	$5 > 100$
	set	68	75	362	$5 > 100$
	size	18	23	93	$0 > 100$
	increment	100	105	331	$5 > 170$
	sum	93	97	168	$5 > 150$
2	get	75	83	1212	$9 > 150$
	set	68	82	1787	$9 > 150$
	size	18	25	718	$2 > 150$
	increment	100	115	993	$8 > 200$
	sum	93	111	1756	$13 > 200$
3	get	56	90	2075	$23 > 150$
	set	56	86	1256	$16 > 150$
	size	12	25	862	$3 > 150$
	increment	87	167	1831	$26 > 300$
	sum	81	129	1893	$24 > 300$

Table 2: Wall Clock Times (in Microseconds) for MdartsArray with Spinlocks.

caused this jitter. The "size" transaction worst-case time was 93 microseconds compared with 23 microseconds on average. The 70-microsecond difference indicates the granularity of the scheduler interrupt when no task switching is performed.

If one examines the "size" transaction in the two-CPU case, one can see that one of the transactions was timed at 718 microseconds. Clearly, this was a task preemption while the timer was running. It appears that 800 microseconds is a typical execution time for tasks that preempted our transactions. Most probably, the preempting task in this case was an ethernet servicing task run by the operating system. One might wonder why we are encountering preemption delays in transactions that disable preemption during critical sections. The answer is that parts of the transaction outside of the critical section can still be preempted.

We disable preemption (task switching) in critical sections for two primary reasons. First, we want to tightly bound the number of tasks that can enter a spinlock queue for a particular object. If preemption is disabled when tasks enter the queue, no more than one task per CPU can be in the queue. By bounding the queue lengths, the lock objects can make real-time guarantees regarding the maximum delays to acquire the lock. Second, we wish to disable preemption to avoid remote processor blocking if a task ahead of it in the queue is preempted. In general, we are willing to be preempted by local high-priority local tasks (such as the ethernet servicing task), but we want to avoid delays caused by preemptions on a remote processor. By disabling preemption but not disabling interrupts during critical sections, we are exposing transactions to the possibility of being delayed by remote interrupt processing. However, the total interrupt processing utilization for the task scheduler on our system is about 70 microseconds per 16 milliseconds, or 0.0044 utilization. scheduler utilization is probably even less when task switching is disabled, but for the sake of discussion, assume that it is 0.0044. If scheduler interrupts are permitted during critical sections on our three processors, in the worst case this will result in utilization loss of 0.0088 for each processor (each processor may have to delay the amount of time required for handling scheduling interrupts on the other two processors). Although this is a very pessimistic assumption, the utilization loss is still very low.

7.2.1 Transaction Throughput

It is interesting to examine the raw transaction throughput of the array transactions. With one CPU, getting a lock, reading or writing one integer in the array, releasing the lock, and returning the result requires about 75 microseconds. This corresponds to over 13,000 transactions per second. The "sum" transaction (which gets a lock, sums the 10 array elements, releases the lock, and returns the result) requires about 97 microseconds, about

10,000 transactions per second. The "increment" transaction is roughly as fast as the "sum" transaction. The "size" transaction, because it requires no locking, can perform about 40,000 transactions per second per CPU. Table 3 summarizes the average throughputs of Table 2. To put these performance numbers in perspective, prior RTDBSs typically achieve ten to 100 transactions per second. Furthermore, the MDARTS performance numbers reported here correspond to worst-case contention and transaction load conditions.

CPUs	get	set	size	increment	sum
1	13,000	13,000	43,000	9,500	10,000
2	24,000	24,000	80,000	17,000	18,000
3	33,000	35,000	120,000	18,000	23,000

Table 3: Throughput (in Transactions Per Second) for MdartsArray Transactions.

Table 3 shows that for transactions with short critical sections, we achieve nearly linear speedup as processors are added. The "increment" and "sum" transactions do not show as much speedup since they have longer critical sections, and time spent waiting to enter a critical section is unproductive.

The linear speedup of the "size" transactions shows that bus bandwidth is not a limiting factor in these experiments. This is not surprising since our platform's VME bus can support about 5 bus operations per microsecond. The "size" transaction requires 23 microseconds to complete, and it issues only one VME request. Therefore, a CPU running thousands of "size" transactions in a tight loop will consume only one percent of the VME bandwidth. Similar observations apply to the other transactions supported by MdartsArray. The primary factor that limits speedup in our experiments is the mutual exclusion enforced by the locks, not the bus bandwidth. However, on systems with large numbers of CPUs, the bus may become a bottleneck.

"Increment" has a longer critical section since it generates two VME operations per array element (for read/add/write), whereas "sum" generates only one (read/add). Since the critical section of "increment" is one third of its overall execution time, we can never achieve better than a threefold speedup through parallelism. Similarly, the "sum" transaction can be speeded up only by a factor of four. Table 3 supports this conclusion since the speedup is greater for "sum" than "increment" when a third CPU is used.

7.2.2 Worst-case Transaction Times

We have discussed the throughput of our MdartsArray transactions in terms of total transactions per second, because this metric is commonly used to evaluate database system performance. However, hard real-time systems must be designed for worst-case conditions, not average-case. Therefore, it is important to consider the worst-case transaction performance. Clearly, it is difficult to measure the true worst-case transaction performance, because system interrupts and task preemptions greatly increase the wall clock uncertainty of any given transaction. (Note that we measured wall clock times of transactions in Table 2). However, the true measure of transaction performance is actual execution time, not wall clock time. It is execution time that directly affects the utilization and hence the schedulability of tasks that perform transactions (provided priority inversion is bounded). In other words, it is wrong to attribute preempted time to the operation that was preempted. However, time spent spinning in a spinlock queue is execution time that should be attributed to that transaction.

Therefore, spin wait delays attributable to remote system(???) interrupt processing (e.g., when a remote lock holder is interrupted by its CPU scheduler) should be factored into an overall utilization reduction, not charged to each transaction that might encounter such a delay. This is because the maximum cumulative delay associated with waiting during remote interrupts is bounded for a given period of time (assuming interrupt rate and service times are bounded, which is true of any properly-designed real-time system). For example, if 100 transactions are performed during a short period of time in which only one remote interrupt can occur (because the interrupt rate is bounded), then it is unnecessarily pessimistic to charge an interrupt service time to each of the 100 transactions when calculating worst-case database performance. Instead, one remote interrupt service time should be charged to the entire set of 100 transactions. straightforward way to accomplish this is to reduce the available processor utilization on each CPU that uses the database by the sum of interrupt servicing utilizations on other CPUs with which it shares database objects. If this results in too severe a utilization penalty (as may be the case if average interrupt service times are long or interrupt rates are high), then the lock objects should disable interrupts during critical sections. If remote interrupts are accounted for with a utilization deduction, delays due to remote interrupts can be discounted when considering worst-case MDARTS transaction time guarantees.

Since we have eliminated interrupt and preemption-related delays from our worst-case transaction time analysis, the worst-case transaction time becomes a function of the transaction code implementation, characteristics of the hardware platform (e.g., bus bandwidth), the level of concurrency, and the locking protocol used for that transaction. In the single CPU case, the worst-case execution time in our experiments is almost identical to the average-case execution time. This is because without concurrency control delays (with only one CPU, the transaction always acquires the lock), the only significant source of execution time uncertainties is VME bus access time. With three CPUs performing simultaneous bus operations and round-robin scheduling, the delay to access the bus will be less than one microsecond per bus operation (assuming five bus operations per microsecond).

It is important to note that while a CPU is waiting in the spinlock queue, it generates no VME bus operations (it spins on a local control/status register in its VME chip). Therefore, as lock contention increases, bus contention decreases. This effect reduces the worst-case latency attributable to the bus as queue lengths increase. If each CPU performs transactions on different objects, the bus contention is maximized, but none of the transactions will be delayed in the spinlock queues. The net result of these two alternatives is that worst-case transaction performance remains high even under heavy loads.

Table 4 summarizes the worst-case transaction times for our MdartsArray objects. These are estimates rather than actual measurements since preemptions and remote interrupts prevented us from directly measuring worst-case transaction times (as we have defined them). Nevertheless, these estimates should be quite close to the actual worst case (given round-robin bus scheduling and no uninterruptible DMA activity).

Table 4 assumes critical section lengths of 6 microseconds for "get" and "set," transactions 35 microseconds for "increment," and 25 microseconds for "sum." For example, the "get" and "set" transactions' worst case execution times are 75 microseconds plus 6 microseconds per CPU. (We assume that the locking overhead of each transaction is 75 microseconds.) If Table 4 is compared with tables 2 and 3, one can see that the average-case transaction times under the high transaction loads generated by our experiments correlate well with these worst-case estimates.

Note that these worst-case transaction times assume maximum transaction load conditions. Under normal conditions, much less contention would be observed for a given MDARTS object. However, since we are targeting hard real-time systems, we must ensure that our transaction-time guarantees will be valid even under heavy load conditions. It is very significant that the worst-case transaction times are so short. This makes MDARTS suitable for high-speed hard real-time applications. We are able to achieve this high performance under extreme load conditions because of our locking protocols and our approach to concurrent transaction processing across the multiprocessor.

CPUs	get	set	size	increment	sum
worst-case transaction times in microseconds					
	75 + 6C	75 + 6C	25	75 + 35C	75 + 25C
1	81	81	25	110	100
2	87	87	25	145	125
3	93	93	25	180	150
worst-case throughput in transactions-per-second					
1	12,000	12,000	40,000	9,000	10,000
2	23,000	23,000	80,000	14,000	16,000
3	32,000	32,000	120,000	17,000	20,000

Table 4: Estimated Worst-case Performance for MdartsArray Transactions.

7.3 Remote Access

We have presented the performance of direct shared-memory access by two MDARTS classes in the above subsections. However, MDARTS also supports remote access through RPC calls to the Shared Data Manager server. Although we cannot analyze the worst-case performance of our RPC-based transactions (because the networking protocols we use do not provide guarantees), we can measure typical performance of RPC-based MDARTS transactions on our platform. As we shall see, the communication delays completely dominate the transaction times for short transactions such as supported by MdartsArray. Figure 15 illustrates the tasks and communication involved in our remote experiments. In this case, the MDARTS objects used by the Experiment objects (labeled RO) are RemoteMdarts objects that use RPC to forward transaction requests on to the SDM server. The SDM performs the transactions and returns the results to the application tasks. When the SDM processes a transaction request, it performs that transaction using the same shared memory transaction methods that the local object supports. Therefore, each "get" or "set" transaction requires only about 15 microseconds of execution time in the SDM. However, the overhead implicit in the client-server architecture reduces performance by three orders of magnitude.

The CPU boards we used in our experiments could use two networks. One was the campus-wide ethernet. The other was a separate network implemented by the VxWorks operating system across the VME bus backplane. We ran experiments with remote objects on both of these networks. Our first set of experiments used the campus ethernet. In these experiments, we ran the SDM on only one of the three CPUs. Experiments involving one or two CPUs did not run experiment tasks on the CPU that hosted the SDM. However,

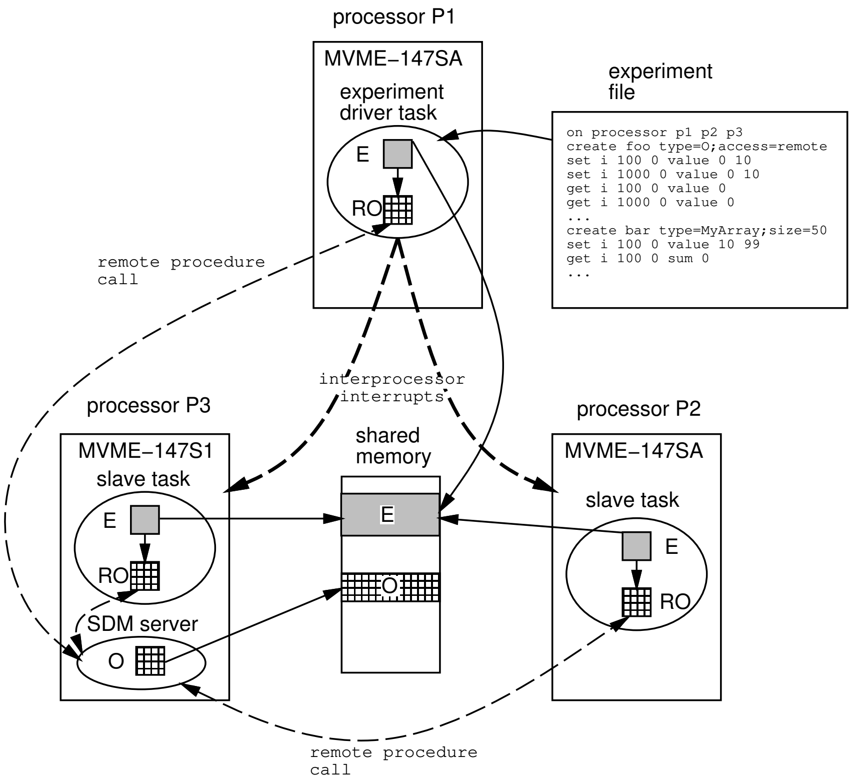

Figure 15: MDARTS experiments with remote objects.

experiments with three CPUs necessarily ran one of the experiment tasks on the same CPU as the SDM. Table 5 presents the results of running the remote experiments across the campus ethernet. The transaction times for "get" and "set" transactions were equal, so the transaction type is not labeled in Table 5. Note that the average throughput increases nearly linearly as the CPU number increases. However, the average transaction times are about 32,000 microseconds for transactions across the ethernet, whereas the average transaction times were only about 15 microseconds when direct shared-memory access was used.

Interestingly, the average performance is actually better when there are two clients (the two-CPU case). This is probably due to the server task on CPU 3 being more often ready to service requests when they are more frequent. If requests are relatively infrequent, the SDM server will block between requests. Each request is then delayed by the time required to schedule and start the server task running again. If the server task has just finished serving a request from CPU 1 when a request from CPU 2 arrives, it can immediately service the new request thus reduce the blocking and task switching overhead.

CPUs	best	average	worst	average throughput (TPS)
1	31,400	33,000	80,000	30.3
2	9,000	31,000	83,000	64.5
3	14,900	34,800	100,000	86.2

Table 5: Remote Wall Clock Times (in Microseconds) for MdartsInt Across Ethernet.

One would think that substantial performance improvement could be gained by using the VME backplane network rather than the ethernet wire. To investigate this, we ran an additional set of experiments using this alternative network and our remote access objects. The results of these experiments are reported in Table 6. With one and two CPUs, the average throughput using the backplane network is approximately twice that of the campus ethernet. Furthermore, the worst-case times are very close to the average-case. On the campus ethernet, worst-case times were two to three times the average case.

However, when the third CPU is used, the overall throughput actually declines. This is because the third CPU was the host of the SDM server task. When the experiment task on that CPU was invoked, the VxWorks task scheduler began context switching between the experiment task and the SDM. This caused a significant decrease in the overall transaction throughput. To investigate this effect further, we ran an additional experiment using only the CPU with the SDM task. The result of this experiment is the 1^* row of Table 6. Running the client and server on the same CPU yielded approximately the same throughput as using the campus ethernet, about half of the throughput when the client and server were on different CPUs.

We did not observe this large throughput decline when we added a client to CPU 3 using the campus ethernet (in Table 5). This is probably due to the hardware support of the LANCE ethernet chip. This chip could perform many of the duties that were delegated to the CPU when the backplane was used. Therefore, in the ethernet case, response times were limited by communication latencies, and in the backplane case, response times were limited by CPU power. When three client tasks were used, neither method showed a clear advantage over the other.

CPUs	best	average	worst	average throughput (TPS)
1	14,700	16,700	17,000	59.9
2	9,600	16,600	18,000	120
3	14,900	33,300	77,500	90
1*	27,700	33,300	83,300	30

Table 6: Remote Wall Clock Times (in Microseconds) for MdartsInt Across VME Backplane.

8 Conclusions and Future Work

8.1 Research Contributions

MDARTS makes many important contributions to the fields of RTDBS and real-time object-oriented systems. First and foremost, MDARTS is an actual implementation of a hard real-time database system with very high performance. Prior RTDBS prototypes are designed only for soft real-time systems, and their performance is insufficient for applications with sub-millisecond transaction deadlines. By moving transaction processing into application tasks, using spinlock queues for concurrency control, MDARTS achieves high predictability and two to three orders of magnitude performance improvement over prior RTDBSs for memory-based transactions typical of machine controllers.

Database systems and distributed object-oriented systems are almost universally implemented using a client-server architecture. We have shown why this architecture implies system-related overhead that can drastically degrade real-time performance, especially when individual transaction times rather than aggregate throughput are considered. The primary reason MDARTS is so much faster than prior RTDBSs is that we avoid the client-server architecture for hard real-time transactions. Note that if high-performance remote procedure calls are available with worst-case latency bounds, the client-server architecture becomes more competitive with the direct shared-memory approach. MDARTS defines a framework for expressing performance characteristics and requirements, and the database class designer is free to implement MDARTS objects using whatever techniques are appropriate.

MDARTS can run on both uniprocessors and multiprocessors. On shared-memory multiprocessors, MDARTS is able to fully exploit the parallelism available in the hardware with minimal overhead. Prior RTDBSs and object-oriented systems for multiprocessors either incur serial bottlenecks in server processes, or they duplicate data across the processors and incur substantial overhead maintaining data consistency.

Another key contribution of MDARTS is the way it uses application-specified timing and semantic constraints to customize the selection of data management classes. MDARTS allows applications to make their requirements explicit in the contracts processed by the object constructors. By providing a mechanism for customizing object creation according to application needs, MDARTS can enhance performance without requiring application programmers to know exactly which database class to use.

MDARTS also provides finer granularity of method timing specification than prior real-time object-oriented systems since each transaction method can have multiple timing records corresponding to different parameters. Furthermore, MDARTS includes support for automatically measuring method execution times, scaling performance to benchmarks performed on the computing platform, and estimating worst-case resource sharing delays at runtime. Prior real-time object-oriented systems require application developers to specify method execution times by hand.

8.2 Future Directions

There are many areas in which MDARTS could be enhanced. First, it would be very useful to develop interfaces to file-based database systems. Real-time transactions could prefetch and cache persistent information in memory to avoid long I/O delays during transaction execution. Main-memory database researchers have investigated many algorithms for transaction logging and recovery in memory-based systems. It would be interesting to determine which, if any, of these methods could be used in MDARTS to make its memory-based objects persistent.

It would be interesting to experiment with multiversion concurrency control techniques to eliminate blocking delays for MDARTS transactions. Our spinlock queues limit multiprocessor speedups in proportion to the number of processors and the lengths of the critical sections (assuming that all processors access the same object simultaneously). Multiversion concurrency control can improve transaction time guarantees for database objects that are likely to be used by many tasks across large numbers of CPUs. Since MDARTS permits each database class to use its own concurrency control strategy, the multiversion technique could be applied judiciously to those objects that are good candidates for that approach.

Finally, it would be very useful to develop a set of basic data management classes for real-time control systems and make MDARTS available to industrial and academic developers of machine controllers.

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. Adelberg, B. Kao, and H. Garcia-Molina, "Overview of the STanford Real-time Information Processor (STRIP)," *SIGMOD Record*, vol. 25, no. 1, pp. 34–37, March 1996.

[3] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut, "Prisma/db: A parallel, main memory relational dbms," *IEEE Trans. Knowledge and Data Engineering*, vol. 4, no. 6, pp. 541–554, December 1992.

[4] A. Attoui and M. Schneider, "An object oriented model for parallel and reactive systems," in *Proc. Real-Time Systems Symposium*, pp. 84–93, December 1991.

[5] E. Bensley, P. Krupp, R. A. Sigel, M. Squadrito, B. Thuraisingham, and T. Wheeler, "Object-Oriented Implementation of an Infrastructure and Data Manager for Real-Time Command and Control Systems," in *Proc. the Workshop on Object-Oriented Real-Time Dependable Systems*, pp.201–209, February 1996.

[6] 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.

[7] M. J. Carey, R. Jauhari, and M. Livny, "Priority in dbms resource scheduling," in *Proc. Int'l Conf. on Very Large Data Bases*, pp. 397–410, 1989.

[8] S. C. Cheng and J. A. Stankovic, "Scheduling algorithms for hard real-time systems: A brief survey," in *IEEE Tutorial: Hard Real-Time Systems*, J. Stankovic and K. Ramamritham, editors, IEEE Computer Society Press, 1988.

[9] L. B. C. DiPippo and V. F. Wolfe, "Object-based semantic real-time concurrency control," in *Proc. Real-Time Systems Symposium*, pp. 87–96, December 1993.

[10] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Second Edition, Addison-Wesley, 1994.

[11] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, December 1992.

[12] P. Gopinath, R. Ramnath, and K. Schwan, "Data base design for real-time adaptations," *J. Systems Software*, vol. 17, no. 1, pp. 155–167, 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, and D. Towsley, "Experimental evaluation of real-time optimistic concurrency control schemes," in *Proc. Int'l Conf. on Very Large Data Bases*, pp. 35–46, September 1991.

[15] 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.

[16] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "An object-oriented real-time programming language," *IEEE Computer*, vol. 25, no. 10, pp. 66–73, October 1992.

[17] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dali: a high performance main memory storage manager," in *Proc. Int'l Conf. on Very Large Databases*, 1994.

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

[19] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," in *Proc. 11-th Int'l Conf. on Distributed Comput. Systems*, pp. 300–307, May 1991. (An improved version appeared in the *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, Oct. 1994.

[20] T.-W. Kuo and A. K. Mok, "Ssp: a semantics-based protocol for real-time data access," in *Proc. Real-Time Systems Symposium*, pp. 76–86, December 1993.

[21] J. Lee and S. H. Son, "Using dynamic adjustment of serialization order for real-time database systems," in *Proc. Real-Time Systems Symposium*, pp. 66–75, December 1993.

[22] T. J. Lehman, E. J. Shekita, and L.-F. Cabrera, "An evaluation of starburst's memory resident storage component," *IEEE Trans. Knowledge and Data Engineering*, vol. 4, no. 6, pp. 555–565, December 1992.

[23] M. Lehr, Y. Kim, and S. H. Son, "StarBase: A Firm Real-Time Database Manager for Time-Critical Applications," in *Proc. 7th Euromicro Workshop on Real-Time Systems*, pp 317-322, 1995.

[24] S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala, "The MARUTI hard real-time operating system," *ACM Operating System Review*, vol. 23, no. 3, , June 1989.

[25] K. Li and J. F. Naughton, "Multiprocessor main memory transaction processing," in *Proc. IEEE Int'l Symp. on Databases in Parallel and Distributed Systems*, pp. 177–187, December 1988.

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

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[28] V. M. Nirkhe, S. K. Tripathi, and A. K. Agrawala, "Language support for the MARUTI real-time system," in *Proc. Real-Time Systems Symposium*, pp. 257–266, December 1990.

[29] S. Nishio, K. F. Li, and E. G. Manning, "A time-out based resilient token transfer algorithm for mutual exclusion in computer networks," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 386–393. IEEE, June 1989.

[30] 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.

[31] 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.

[32] R. Rajkumar, *SYNCHRONIZATION IN REAL-TIME SYSTEMS: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

[33] K. Ramamritham, "Real-time databases," *Int'l Journal of Distributed and Parallel Databases*, 1992. (Invited Paper)

[34] K. Schwan, P. Gopinath, and W. Bo, "CHAOS-kernel support for objects in the real-time domain," *IEEE Trans. Computers*, vol. C-36, no. 8, pp. 904–916, August 1987.

[35] 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.

[36] M. Singhal, "A fully-distributed approach to concurrency control in replicated database systems," in *IEEE Proc. Int'l. Computer Software and Applications Conference*, pp. 353–360, 1988.

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

[38] P. Sleat and P. Osmon, "A methodology for real-time database system construction," in *Proc. Int. Conf. on Software Engineering for Real Time Systems*, pp. 233–238, September 1991.

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

[40] 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.

[41] 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.

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

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

[44] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," Technical Report CMU-RI-TR-93-11, Carnegie Mellon University, July 1993.

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

[46] H. Tokuda and C. Mercer, "Arts: A distributed real-time kernel," *SIGOPS*, vol. 23, no. 3, , 1989.

[47] O. Ulusoy and G. G. Belford, "Real-time lock-based concurrency control in distributed database systems," in *Proc. Int. Conf. on Distributed Computer Systems*, pp. 136–143, 1992.

[48] O. Ulusoy and A. Buchmann, "Exploiting Main Memory DBMS Features to Improve Real-Time Concurrency Control Protocols," *SIGMOD Record*, vol. 25, no. 1, pp. 23–25, March 1996.

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

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

[51] V. F. Wolfe, L. C. DiPippo, J. J. Prichard, J. M. Peckham, and P. Fortier, "The Design of Real-Time Extensions to the Open Object-Oriented Database System," in *Proc. the Workshop on Object-Oriented Real-Time Dependable Systems*, October 1994.