AN OBJECT-ORIENTED REAL-TIME DATABASE SYSTEM FOR MULTIPROCESSORS.

by

Victor Bradley Lortz

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:
- Professor Kang G. Shin, Chair
- Professor Toby J. Teorey
- Assistant Professor Elke A. Rundensteiner
- Professor Galip Ulsoy
- Associate Research Scientist Chinya V. Ravishankar

ABSTRACT

AN OBJECT-ORIENTED REAL-TIME DATABASE SYSTEM FOR MULTIPROCESSORS.

by
Victor Bradley Lortz

Chair: Kang G. Shin

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 software system called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). MDARTS avoids the client-server overhead of most prior real-time database systems and object-oriented real-time systems by moving transaction execution into application tasks. By eliminating these sources of overhead and focusing on basic data management services for hard real-time systems (data sharing, serializable transactions, and multiprocessor support), our MDARTS prototype provides guaranteed transaction times approximately three orders of magnitude faster than prior real-time database systems. 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 database object initialization time and attempts to construct objects that are compatible with the requirements. MDARTS supports both local shared-memory transactions and remote transactions that use remote procedure calls. Except for variations in transaction time guarantees, the locations and implementations of MDARTS objects are transparent to applications. MDARTS provides a C++ interface rather than a query language interface. Our MDARTS prototype runs on VME-based multiprocessors and Sun workstations, and we have used MDARTS to implement a controller for an actual manufacturing machine.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER 1
INTRODUCTION

1.1 Motivation

Real-time systems are an increasingly important class of computer applications. A computing system is considered real-time if it has deadlines associated with its computations. Examples of real-time systems include advanced manufacturing systems, air traffic control systems, telecommunications systems, nuclear reactor controllers, and "smart" weapons systems. In a hard real-time system, such as a machine controller, missing a single deadline could result in catastrophic failure. Soft real-time systems can tolerate occasionally missing deadlines. Although the techniques presented in this dissertation could be used for soft real-time systems, we focus primarily on hard real-time systems.

As real-time applications become more complex and 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 conventional database systems (i.e., file- or memory-based non-real-time database systems) are not designed to provide the performance levels or predictability 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 requires worst-case 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. In the future, we expect conventional database performance to improve, but by then high-speed real-time applications will require even greater performance.

It is possible to improve database performance by keeping the database in memory and avoiding disk I/O during transaction processing [23]. However, conventional main memory databases are designed to maximize average throughput, not to minimize individual transaction times. Typical average transactions times for simple transactions in main memory databases (600 milliseconds for TPK [45], about 69 milliseconds for the main memory version of Starburst with concurrency control disabled [42], over 100 milliseconds for PRISMA/DB

[4]) are much too slow for high-speed 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 reported in [13, 41, 34, 70, 77, 88], none provides hard real-time guarantees, and none has average transaction times of less than 100 milliseconds. Thus, these 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 real-time systems. To provide greater flexibility and to manage the complexity of future real-time applications, better real-time data management technology is needed.

This dissertation describes the design and implementation of a 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++, an object-oriented programming language [85]. Applications using MDARTS can specify real-time requirements for transactions in the declarations of their database objects, and they can query database objects to determine real-time characteristics prior to performing transactions.

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. For example, most database systems provide interpreters for ad hoc queries expressed in a database language such as SQL. Machine control systems have no need for ad hoc queries. In this application domain, the raw performance of the database is much more important than report-generating features or the ability to provide multiple views of the data. A database purist may, upon reading this dissertation, conclude that we have not created a database system at all. That would be a perfectly valid interpretation, given a fairly narrow definition of a database system. However, 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.

We characterize the real-time performance of MDARTS with the term *transaction time*. An MDARTS transaction time is composed of two components. The first component represents local task execution time required to perform a transaction. The second component represents blocking delays caused by a transaction 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. The sum of the two transaction time components corresponds to the *response time* metric commonly used in conventional database systems.

MDARTS uses semantic information supplied by application tasks to adjust its data management services during object initialization and to reserve sufficient resources to meet the requirements (or to signal a problem, if the requirements cannot be met). For example, an application might specify that only one task will be updating a particular database object. Given this semantic information, MDARTS can choose a database class that is optimized for single-writer concurrency control.

Transactions with guaranteed transaction times are executed directly by application tasks rather than by separate database servers. In this way, MDARTS avoids the system overhead of context switching and inter-process communication implicit in most database systems. Our prototype implementation can guarantee transaction times of less than 100 microseconds for simple, memory-based transactions typical of machine controllers. In this context, the transaction times consist entirely of local task execution time (the blocking times are zero since busy waiting is used for concurrency control). These performance levels are achieved using commercially available multiprocessor hardware and a commercial real-time operating system kernel (20 MHz 68030 processors running VxWorks). These processors are fairly slow by today's standards, and with modern processors performance could be improved by at least an order of magnitude. MDARTS also provides network access to remote objects through proxy objects that forward requests to database servers via remote procedure calls (RPC). The relatively slow RPC transactions provided by MDARTS servers do not delay the fast memory-based transactions executed directly by application tasks on the multiprocessor. Except for variations in transaction time guarantees, the locations and implementations of MDARTS objects are transparent to applications.

1.2 Background

1.2.1 Real-Time Computing

A computation is defined as real-time if its correctness depends 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. Examples of soft real-time systems include on-line transaction systems such as program trading or airline reservation systems. Hard real-time systems have strict deadlines; catastrophic failure can occur if even one of these deadlines is missed. For example, a manufacturing machine controller may recompute its control signals every 1 or 2 milliseconds. Failure to meet this deadline could cause the machine to become unstable and malfunction, possibly with dire consequences.

A common misconception about real-time computing is that it is equivalent to high-speed computing [82]. Actually, there are fundamental differences between the two. Whereas high-speed computing refers to average performance levels, real-time computing requires absolute performance levels. To guard against failure, hard real-time systems are typically designed using worst-case assumptions about all operations. If the average case differs significantly from the worst case, this will lead to severe underutilization of resources. Furthermore, real-time systems must keep up with external events and changes in the system being monitored. Conventional software is not constrained to respond as quickly or as predictably to external events. Because of these fundamental differences, algorithms and software architectures suitable for high-speed computing often are inadequate for real-time systems.

1.2.2 Real-Time Data Management

Historically, real-time system designers have taken an ad hoc approach to data management. Shared data are sometimes passed between tasks via message queues and sometimes kept at predetermined locations in shared memory. For simple systems with limited inter-task data coupling, an ad hoc approach to data sharing suffices. However, as data volume increases and software becomes more complex, ad hoc data management becomes inadequate.

The Next Generation Workstation/Machine Controller (NGC) for automated factories is representative of the class of complex, distributed real-time architectures that requires data management services [2, 51]. The NGC is a software architecture specification for advanced cell-level machine tool controllers. In this context, cell-level refers to a manufacturing system workcell, which is a factory component that might contain one or more robots, a computer-controlled milling machine, etc. The NGC architecture was designed for high-performance computing platforms such as VME-based shared-memory multiprocessors. An NGC-compatible controller consists of multiple hardware and software components possibly supplied by different vendors. Decomposition of NGC software components into separate tasks on multiple CPUs complicates data management. Shared data must be made accessible to, and used consistently by, all tasks that access them. It is also necessary to control concurrent access to prevent data corruption. Clearly, these requirements match the traditional capabilities of database management systems. Therefore, the NGC specifies a database called the Information Base Subsystem (IBS) to provide data sharing and communication between different software modules. For real-time tasks, the IBS should also provide strict transaction-time guarantees. Interestingly, the NGC does not provide any specification of transaction time requirements for the IBS. This is a serious limitation in the NGC architecture.

Real-time systems such as the NGC require database services, but they do not require all features provided by conventional database systems. Data values representing the state of the controlled system can change very rapidly, sometimes thousands of times per second. It is too expensive to provide traditional database transaction semantics in this context. In particular, data persistence and failure recoverability are not needed for data corresponding to transient sensor readings. Furthermore, often only one task will be performing updates to a given data value. Therefore, it is possible to optimize the concurrency control strategy to match the semantics of the application.

1.2.3 Real-Time Database Research

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. Prior RTDBS research investigates three primary strategies for improving the performance and predictability of database transactions: 1) use memory-based databases, 2) schedule transactions according to task priorities and/or deadlines, and 3) reduce delays and uncertainties associated with concurrency control. To this list, we add: 4) avoid the overhead associated with a client-server architecture, and 5) make maximum use of parallelism on multiprocessor systems.

Main Memory Databases

Some RTDBS researchers propose using main memory databases to eliminate blocking time uncertainties associated with disk I/O during database transactions [71, 80]. There has also been significant interest recently in using main memory databases to increase performance for conventional transaction processing systems [4, 23, 42, 45]. Garcia-Molina and Salem present a nice overview of main memory database research in [23]. The primary limitation of conventional main memory database systems, from the perspective of hard real-time applications, is that these database systems are designed to maximize average transaction throughput rather than to minimize worst-case individual transaction times. For example, the TPK multiprocessor main memory database system reported in [45] achieves an average throughput of over 1,300 transactions per second on a multiprocessor with five one-MIPS processors.

If TPK's 1,300 transactions per second corresponded to a guaranteed transaction time of one millisecond, it would be sufficient for many hard real-time systems (especially since much faster processors are now available). However, TPK achieves this average performance level by processing 650 transactions as a group. The database does not commit an individual transaction until the entire group is processed. The average execution time to process a group of 650 transactions is about 400 milliseconds (these are fairly simple transactions). Communication and context-switch overhead further delay the response times of individual transactions so that the average response time is about 600 milliseconds. The authors report that 99% of transactions in their experiments completed within one second. This implies that some transactions had response times over one second (more than 1,000 times slower than the average throughput implies). Therefore, this system is not well-suited to the needs of high-speed hard real-time systems. Like TPK, MDARTS is designed for shared-memory multiprocessors. However, MDARTS avoids the system overhead that led the TPK designers to process transactions in groups. MDARTS transactions that directly access data in shared memory require no context switching or inter-process communication. Therefore, MDARTS transactions need not be batched together to achieve high throughput and fast individual transaction times.

Clearly, using main memory does not in itself yield a real-time database system. The database system must be designed specifically for the needs of real-time applications. MDARTS combines main memory with parallel transaction execution performed by application tasks rather than database server tasks. Data integrity and concurrency control are supported by MDARTS since all data access is through the transaction methods exported by the database objects. This transaction execution model maximizes the performance benefits of using main memory because it eliminates the overhead implicit in client-server architectures. None of the prior work in RTDBSs considers the possibility of application tasks executing transactions themselves.

Transaction and I/O Scheduling

Several researchers have investigated transaction and I/O scheduling algorithms that support different real-time needs and priorities [1, 12, 62, 83, 76]. Some commercial database systems support priority-based transaction scheduling [25, 64]. By servicing high-priority tasks first, the database can provide faster and more predictable performance for transactions submitted by high-priority tasks. In this case, low-priority tasks experience degraded performance. File-based databases in particular can benefit from transaction and I/O scheduling according to priorities. Memory-based databases benefit less because the scheduling overhead may become a significant percentage of the transaction processing time. One of the difficulties with database transaction scheduling is that interactions with the operating system's task scheduler must be considered.

Serializability and Concurrency Control

Locks are often used in database systems to guarantee serializability. Serializability is a transaction property that means the effect of running interleaved concurrent transactions is equivalent to running them in some serial order. Serializability is the most widely accepted criterion for correct concurrency control. It is useful since it permits transactions to be designed as if they had exclusive access to the database. A major source of performance uncertainty in conventional databases is the potential for transaction delays or aborts when concurrent transactions compete for locks. In a real-time system, such delays could cause critical tasks to miss deadlines. This problem and its contributing factors have been the focus of most RTDBS research.

The most popular locking protocol in conventional database systems is two-phase locking (2PL). This algorithm is easy to implement, and it guarantees serializability. However, two-phase locking can lead to unpredictable delays in transaction processing since 2PL does not bound the worst-case time required to obtain the locks and proceed with the transaction. For conventional applications, these delays are usually tolerable, but unbounded delays are a serious problem in real-time systems. Therefore, a significant amount of work has been done to evaluate alternative concurrency control strategies for real-time synchronization. Nishio *et al.* advocate a cautious transaction scheduling approach that checks for non-serializable execution as it executes transactions [56]. This cautious approach performs better than 2PL, and it never aborts or rolls back transactions to achieve concurrency control. Haritsa *et al.* [32] and Lee and Son [41] advocate various optimistic concurrency control protocols which proceed with transactions and only check for serialization problems at commit time. Huang *et al.* [34], however, dispute some of the conclusions of Haritsa *et al.* and observe that overhead associated with implementing optimistic concurrency control can reduce its performance advantages.

Some researchers have investigated concurrency control algorithms for distributed real-time databases. Consistency across replicated data objects can be maintained through transaction timestamps [70, 79] or symmetric updates [71]. Priority inheritance protocols can also be used to reflect task priorities in distributed transaction execution [67].

Some researchers suggest that serializability should not be used as the primary correctness criterion for real-time concurrency control [12, 21, 56]. Lin [48, 49] suggests that, for real-time applications, data inconsistent with the external world can be worse than internally inconsistent data. He calls the correspondence between a database and the state of the external world "external consistency." External consistency is lost if the database cannot process transactions fast enough to keep up with changes in the world. Lin does not explain how a database system should choose trade-offs between internal and external consistency. Clearly, these trade-offs would depend heavily on the particular application. Kuo and Mok [40] introduce a correctness criterion for concurrent transactions which permits unserialized transactions if the data values read and generated by those transactions are sufficiently similar. Epsilon serializability [61] is another alternative to traditional serializability proposed for real-time transactions. The problem with relaxing consistency constraints such as serializability is that it can be more difficult to demonstrate the correctness of transactions.

We agree with Graham [28, 29], who argues that serializability is indispensable as a correctness criterion for concurrent transactions. Furthermore, serializability is not necessarily expensive to achieve. In [29], Graham presents some techniques for verifying the serializability of transactions by analyzing the read and write operations of concurrent transactions. The TPK main memory database system provides serializability without locking by executing transactions serially in a single database server task. In main memory databases, it is common to achieve serializability by simply using serial transaction execution [23]. On a multiprocessor, this is usually accomplished with a database server that serially executes client transaction requests. Disk-based databases cannot afford to use serial transaction scheduling since all transactions would be delayed during I/O operations.

Synchronization delays caused by locking can be reduced if the frequency of locking conflicts is reduced. One approach to reducing locking conflicts is to adjust the lock granularity to lock only data that are affected by each transaction [6, 66, 79]. Reducing lock granularity increases space overhead for locking, and it can degrade performance if many locks must be acquired to perform a transaction. Therefore, the locking granularity should be tuned to transaction characteristics. Semantic and object-based concurrency control protocols extend this idea by characterizing which transactions conflict and thus require serialization [7, 21, 40]. Another approach to reducing locking conflicts is to design transaction protocols that use data versioning to avoid locking altogether [79, 86, 90, 91]. MDARTS objects can use similar techniques in providing concurrency control for their transaction methods. MDARTS does not dictate concurrency control policies, but the MDARTS object-oriented approach facilitates the use of semantic and object-based concurrency control.

Concurrency control in real-time systems can lead to a problem called "priority inversion." Priority inversion occurs when a high-priority task is forced to wait for a lock held by a lower-priority task. If the lower-priority task is preempted by a medium-priority task before it can release the lock, the duration of the priority inversion can become unbounded. An unbounded priority inversion can lead to indeterminate delays in transaction processing, which can cause the high-priority task to miss its deadline. Priority inversion is to some extent unavoidable if resources must be shared, so most researchers have investigated ways to place an upper bound on the duration of priority inversions. One approach to priority inversion is to abort lower priority transactions that conflict with higher priority ones. An alternative is to bound priority inversions using various priority inheritance protocols which temporarily boost the priority of tasks holding locks if they conflict with higher priority tasks [58, 60, 67, 76]. This temporary priority boost helps lower-priority tasks complete transactions and release their locks.

Since the best approach to real-time concurrency control often depends upon the particular application, some researchers advocate hybrid protocols that borrow features of several earlier strategies to perform acceptably for a wider range of applications [12, 32, 35, 60, 78]. For example, Huang *et al.* [35] advocate a hybrid approach in which tasks inherit higher priorities only if they are close to committing. Otherwise, if higher-priority tasks require locks held by low-priority tasks, the lower priority transactions are aborted.

Temporal Databases and Time/Precision Tradeoffs

Some researchers have investigated new approaches to database management based on semantic differences between real-time databases and conventional databases. For instance, a radar tracking application may need to correlate multiple observations that are retrieved from the database. Thus, temporal consistency may be a semantic requirement for a real-time database. Temporal consistency could be achieved by adding constraints to transaction scheduling so that time correlation of data values is maintained [31, 62]. No current schedulers employ this technique, but Liu and Song [81] use temporal correlation as a criterion to evaluate scheduling algorithms. Smith and Liu [75] suggest that real-time databases could return approximate values when precise values cannot be computed before the deadline.

This is an application of the imprecise computation idea of Lin *et al.* [47].

MDARTS does not directly address temporal consistency issues, but it is possible to design data management classes within the MDARTS framework that provide temporal consistency. Similarly, imprecise computation techniques can be built into MDARTS objects if necessary for a particular application.

A Side Note on the Client-Server Architecture

Figure 1.1 illustrates the types of overhead implicit in client-server architectures. We show both multiprocessor and uniprocessor examples. Figure 1.1 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 section **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.

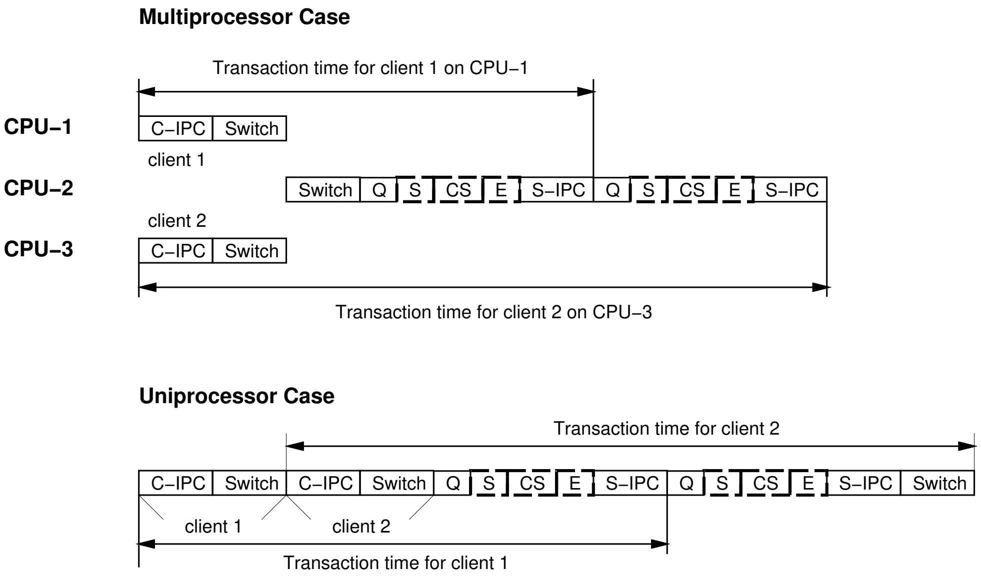


Figure 1.1: Real-time performance implications of the client-server architecture.

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. On a uniprocessor, more context switches are generated as the CPU switches execution from clients to servers. Note that 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.

Some researchers have investigated ways to reduce RPC (remote procedure call) overhead. The x-kernel is an operating system designed to efficiently support multiple RPC protocols [57]. X-kernel RPC overhead is four milliseconds for Sun RPC and 1.7 milliseconds for Sprite RPC on a 2 MIPS processor and a lightly loaded ethernet. Local (same machine) RPCs can be made much faster, depending on how they are implemented in the operating system kernel. By carefully minimizing data copying and using shared memory, Bershad was able to reduce worst-case overhead for local RPCs on DEC Firefly multiprocessors to about 150 microseconds [9]. Bershad's technique has the advantage of propagating the execution priority of the client to the server (the client task is mapped into the server's address space, and it effectively becomes an instance of the server for the duration of the RPC). To accomplish this trick, Bershad relied on several platform-specific features of the Firefly and Taos, its multiprocessor operating system.

With faster processors even better performance can be achieved. For example, by manipulating the virtual memory hardware of the 486 processor and carefully optimizing interactions with the system scheduler, Liedtke was able to reduce worst-case local RPC overhead for short messages (passed in registers) on a 50 MHz 486 uniprocessor to about twelve microseconds (twenty machine instructions) [46]. This represents about an order of magnitude speedup over Mach RPC on the same platform.

Although inter-process communication latencies can be reduced through hardware support and clever optimizations, they can never be completely eliminated. Furthermore, even if RPC overhead could be eliminated, each object's server creates a serial bottleneck that limits parallelism on a multiprocessor (assuming the worst-case scenario in which tasks on multiple CPUs submit simultaneous requests to the same object). This point leads us to another issue: how many objects should be represented by each server? If there is one server per object, the system overhead associated with the server processes discourages designs that use large numbers of objects. If there is one server for multiple objects, each server becomes even more of a bottleneck on the multiprocessor. MDARTS permits application tasks to use large numbers of objects without creating proportionately many server tasks or creating serial bottlenecks.

On our implementation platform, round-trip RPCs consumed several milliseconds between CPUs on the local multiprocessor (we used Sun RPC). The special techniques for minimizing context switch overhead in RPCs described in [9, 46] were not available to us in the commercial real-time operating system we used. Therefore, the theoretical limits of RPC overhead was a moot issue for us. On our platform, RPC was expensive. Instead of trying to reduce RPC overhead, which would have been difficult and would not have been portable, we decided to avoid the client-server architecture for local data sharing.

Figure 1.2 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 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

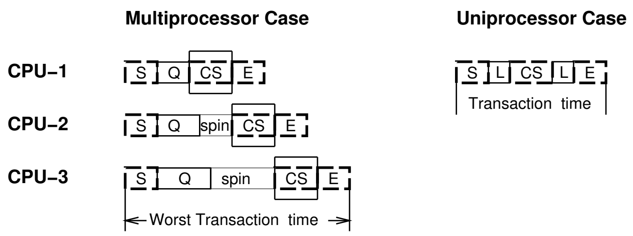


Figure 1.2: Transaction times if clients perform transaction execution.

sections that perform memory-based operations.

As we have said, context-switch and inter-process communication overhead is often much larger than the actual execution times of transactions. Therefore, 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. Pipelining and batch transaction execution can amortize context-switch overhead and communication latencies on parallel machines to achieve high average throughput. However, these techniques do not improve end-to-end response times of individual transactions, a key requirement for real-time applications. 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. Chapter 6 presents these experimental results.

Object-Oriented Approaches to Real-Time Data Management

Object-oriented technology has many features that are particularly useful for developing database systems. A set of basic data management capabilities can be designed into base classes that define a common programming interface and implement certain core functionality. Customized data management classes can then be derived from these base classes, overriding the base class methods where appropriate to extend their capabilities. Because the inheritance mechanism enforces interface compatibility, software developed using the base class methods can safely use objects of derived classes without knowing their exact class. This in turn yields better extensibility and modularity of design. With inheritance and polymorphism, it is very easy to provide data transparency, which means the location and implementation of the data is hidden from the software that uses it. Data transparency is commonly provided by database systems, but object-oriented languages enable efficient implementation of this capability. Object-oriented systems also support the encapsulation of both data and methods (functions) within objects. Data and method encapsulation facilitates the mapping of software objects to physical objects and mechanisms that have internal state. By associating a computation with the context of an object's encapsulated state, higher-level application software need not explicitly maintain implementation-specific state information.

CHAOS [26, 27, 65], Maruti [44, 54], and ARTS [87] provide support for real-time objects at the kernel level of an operating system. ARTS supports real-time objects with lightweight threads and multiple task scheduling policies. A major emphasis in ARTS is the provision of exception handling if deadlines are violated. Maruti guarantees hard real-time deadlines by verifying the schedulability of accepted service requests during a pre-scheduling phase. Maruti and MDARTS share the philosophy of guaranteeing deadlines through runtime initialization, but Maruti primarily focuses on scheduling resources. MDARTS provides atomic data objects with transactions that are not explicitly scheduled by the operating system. Thus, MDARTS requires less global task information than Maruti does, and MDARTS is compatible with existing commercial operating systems.

CHAOS enhances predictability and timeliness by reducing OS-related overhead for object method invocations in multiprocessor and distributed real-time applications. CHAOS objects export methods with heterogeneous invocation semantics. These semantics correspond roughly to different RPC semantics (asynchronous send, synchronous send and reply, etc.). Deadline information is passed at runtime as parameters in CHAOS object invocations. The database designer hard-codes execution times for object methods in the object definitions. These execution times are divided into overhead incurred by the client and overhead incurred by the server in servicing each invocation. CHAOS does not provide support for inheritance, so it is object-based rather than object-oriented (as these terms are usually defined). All three of these operating systems provide application-transparent distributed object method invocation.

CHAOS, Maruti, and ARTS are all based entirely on the client-server model of object sharing. Therefore, these systems incur communication latencies, queueing delays, and serial bottlenecks in the object servers. Schwan *et al.* report some performance numbers for a CHAOS implementation in [65]. According to [65], the best-case overhead for invoking CHAOS methods on the client side alone ranges from one to five milliseconds, depending on the semantics of the invocation. Presumably, the worst-case overhead is substantially worse (the paper does not quantify the worst case). Note that these measurements do not include communication latencies or server-side overhead. In ARTS, each method call incurs about one millisecond of overhead for local objects and about eleven milliseconds for method calls across processors. The paper does not say if this overhead is best or worst-case. Note that this overhead does not include the actual transaction processing or concurrency control delays. Therefore, we conclude that neither CHAOS nor ARTS is well-suited to application domains where sub-millisecond transaction times for object methods are required.

The MO2 model combines features of database systems and real-time systems [5]. MO2 supports distributed active objects with per-object read and write servers that execute client requests at the client priorities. Serializability is provided by executing methods serially in an object's write server process. Unlike CHAOS, MO2 supports inheritance. Like CHAOS, MO2 requires application developers to hard-code execution timing information in the definition of object methods. Since each object has associated with it at least two heavyweight processes, the MO2 architecture makes it expensive to create systems with large numbers of objects. Furthermore, MO2 is also client-server based and thus incurs the overhead associated with that architecture.

Stewart *et al.* describe an object-oriented approach to developing hard real-time applications in [84]. The objects, called port-based objects, follow a strict protocol for sharing information across task boundaries. All cooperating tasks share a common global state table, and during each cycle of a control task's execution, a copy of needed global variables is made by each task. The tasks then 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. Suppose two concurrent tasks on different CPUs each copy the same global variable to their local state tables, modify it, and write it back to the global table. Clearly, these transactions will not be serialized. Furthermore, if applications share large amounts of data, it is expensive to copy it between the local and global state tables each cycle.

Ishikawa *et al.* [36] describe a real-time extension of C++ called RTC++. RTC++ active objects define periodic tasks with multiple threads. Object methods can include declarations of their execution times, and these execution times can be used to perform schedulability analysis of task sets. Concurrency control is implemented in the object methods. If mutual exclusion is needed when executing the object methods, the object is a nonpreemptive object, otherwise, it is a preemptive object. Although RTC++ defines language constructs for expressing timing information, the object developer is responsible for deriving this information. This could be a significant burden, and it suffers the additional drawback that execution times are fundamentally platform-dependent. When timing information is hard-coded into the class definition, there is no allowance for performance variability of computing platforms. This is a serious problem, especially since timing information must be updated and all application code recompiled whenever hardware characteristics change. CHAOS and MO2 share this problem. MDARTS has a more flexible approach that provides support for benchmarking execution times and scaling timing estimates to the performance of the execution platform at runtime (see Section 2.7).

DiPippo and Wolfe have developed an extensive object-oriented model for real-time databases called RTSORAC [21]. Their model includes object-based semantic concurrency control, temporal scopes for object methods (temporal scopes specify very fine-grained timing information about each method), temporal consistency constraints, inter-object transaction constraints, and imprecision constraints. Although this model is extremely powerful, the authors do not report any performance or implementation details that would support a fair comparison between RTSORAC and MDARTS. Like the other real-time object-oriented systems, RTSORAC appears to require developers to specify large amounts of timing information by hand. Furthermore, the runtime overhead of supporting such a complex model are likely to be substantial.

1.2.4 Database Transaction Times and Task Scheduling

The ultimate 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. If tasks are not preemptible, general schedulability analysis is computationally intractable (NP-complete or NP-hard). However, if tasks are preemptible, it is possible to efficiently determine their schedulability by applying rate monotonic scheduling theory [50, 59]. Rate monotonic scheduling is an optimal algorithm for static (pre-assigned) task priorities. Various other dynamic priority scheduling protocols have also been studied (earliest due date, least slack time, etc. [15]).

Rate monotonic scheduling is the most popular scheduling method for practical real-time systems, since it is easy to analyze and more robust under overload conditions than dynamic scheduling approaches. Equation 1.1 defines a set of inequalities that, when satisfied, guarantee the schedulability of a set of n tasks using rate monotonic scheduling. Rate monotonic scheduling assigns task execution priorities for periodic tasks according to the lengths of their periods. The shorter the period the higher the priority. The lower-numbered tasks in Equation 1.1 have higher priorities, and C_i, T_i, and B_i represent the computation time, period, and blocking time of task τ_i, respectively. Note that the blocking time B_i

does not include blocking for higher-priority local tasks, since that is part of the normal preemption interval for task τ_i.

$$\forall i, 1 \leq i \leq n \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1) \tag{1.1}$$

To make schedulability guarantees, it is necessary to know in advance the computation times, periods, and blocking times of all application tasks. Usually the periods of tasks are known in advance (in the case of sporadic tasks, one can use the worst-case interarrival times of the task). 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 using the database can make schedulability analysis impossible. MDARTS addresses this problem by making transaction times short and predictable. When transaction times cannot be predicted (as when non-real-time networking protocols are used in transactions), MDARTS reflects these uncertainties by refusing to guarantee its transaction times.

An MDARTS transaction-time guarantee consists of a worst-case bound on the pure local computation time of the transaction and a worst-case bound on the blocking delay of the transaction. We avoid the more common "response time" metric since it is inappropriate when tasks perform transactions themselves. Most RTDBSs consider the scheduling of multiple client transactions by one or more database server processes. In the client-server model, transaction execution time in the client is much less than the wall clock response time of the database server (which corresponds to blocking delays for the client task). Therefore, the response time is used to characterize the database performance. In MDARTS, tasks usually execute transactions themselves. Unless proxy objects are used, the entire transaction time consists of local execution time rather than blocking time. Therefore, one should analyze execution of the database transaction just as one would analyze the execution of an ordinary code sequence in a task. The wall clock response time of a given set of instructions is not a relevant measure, since higher-priority tasks can preempt and make an arbitrarily short code sequence appear to take a long period of time. This delay is already accounted for in the normal preemption interval of that task. In summary, the two components in an MDARTS transaction-time guarantee correspond directly to the C_i and B_i variables in Equation 1.1. These components are those needed to perform schedulability analysis for tasks that use the database. Note that response times reported in the database literature correspond roughly to the B_i components of an MDARTS transaction time.

If dynamic real-time scheduling techniques are used, the guarantees provided by MDARTS can still be useful. In earliest due date scheduling, the high performance and predictability of MDARTS transaction times can reduce the execution time variance of tasks and bound delays to access shared resources. In least slack time scheduling, the predetermined worst-case transaction times can be used to generate more accurate estimates of slack time for tasks that use the database.

1.3 Research Objectives

Most prior RTDBS research focuses on algorithms and protocols for soft real-time databases. Rather than develop new database architectures, most RTDBS researchers have applied ideas originally developed in the real-time computing field (such as real-time scheduling) to enhance the predictability of traditional database systems. In most cases, the fundamental architecture of the database system has not been questioned. This has led most RTDBS research to use database architectures with one or more database servers that service multiple client tasks. The basic properties of this architecture make it difficult to achieve the levels of predictability and performance required by hard real-time systems. Therefore, most prior RTDBS prototypes have been suitable only for soft real-time systems.

There is a need for soft real-time databases for applications such as airline reservation systems and program trading, but the needs of manufacturing machine controllers and other hard real-time systems are quite different. Our primary research objective is to provide data management services suitable for high-speed hard real-time systems with sub-millisecond transaction-time requirements. None of the prior work in RTDBSs addresses this important application domain. Note that we are not suggesting that every database transaction will have the same guaranteed transaction time. Each transaction might have a unique deadline. The key point is that applications can specify deadlines for individual transactions and can determine in advance if these deadlines will be met.

Another important objective in our research is to provide a flexible and convenient mechanism for expressing and evaluating semantic constraints and transaction-time requirements of applications. Although the performance and predictability of the database system are paramount, it is also important for applications to be able to specify their requirements and determine if the database can meet them. MDARTS uses semantic information provided by applications to configure its data management services according to application needs. MDARTS also supports bidirectional information flow concerning the transaction-time characteristics of its data management services. Applications can use this mechanism to determine the worst-case transaction times of transactions before they are even started.

Our final research objective is to implement and evaluate a prototype implementation of MDARTS to determine its performance and suitability for typical hard real-time applications. Our MDARTS prototype is designed to be portable and flexible without sacrificing real-time performance. Particular attention was paid to making the database's application programming interface simple. Many of the advanced object-oriented techniques employed by MDARTS address issues related to using the library for actual applications. We devoted significant efforts in this area since we wanted to make it as easy as possible for our prototype to be adapted for use in industry. For the same reason, we explicitly avoided reliance on custom hardware support. Our prototype uses strictly off-the-shelf hardware components and operating systems. We implemented our prototype in C++ [85]. We chose C++ because of its wide availability, runtime efficiency, compatibility with C, and object-oriented features. To enhance the portability of our implementation, we used only standard features of C++ rather than adding language extensions.

1.4 A Map of the Dissertation

Chapter 2 presents the core design and implementation of MDARTS. This chapter includes a discussion of the MDARTS transaction model and the different implementation approaches for local shared-memory objects and remote objects. It also describes techniques for automatically calibrating transaction performance estimates for database objects.

Chapter 3 discusses the techniques we use to dynamically customize database services according to application semantics. We show how to use software contracts and exemplar-based object construction to simplify an application's view of a complex library and to guarantee certain semantic and timing constraints before transactions are performed.

Chapter 4 analyzes the problem of assigning priorities to real-time tasks as they wait in global semaphore queues in multiprocessors. Prior work on extending uniprocessor semaphore scheduling to multiprocessors uses task execution priorities for global semaphore queue priorities. We show that better schedulability can be achieved if global semaphore queue priorities are assigned independently of the task execution priorities.

Chapter 5 discusses how we used MDARTS in the real-time control of an actual manufacturing machine. This demonstration also illustrates the proxy (remote transaction) capabilities of MDARTS and shows how it can be used to facilitate integration of existing hardware and software components.

Chapter 6 presents performance results of our MDARTS prototype on a stock multiprocessor. These experiments evaluate the performance of MDARTS under extremely heavy load conditions and demonstrate its multiprocessor capabilities.

The dissertation concludes with Chapter 7, which summarizes the contributions and discusses future work.

CHAPTER 2

MDARTS

2.1 MDARTS Overview

MDARTS is radically different from prior work on real-time databases. Most prior work on real-time databases investigates specific transaction and concurrency control protocols that can enhance the performance or predictability of a database system. Very few actual real-time database implementations are described in the literature, and most of those are testbeds designed for studying algorithms rather than for supporting realistic real-time applications [35, 77]. Furthermore, to our knowledge, none of the RTDBS implementations reported in the literature are intended for hard real-time applications such as machine controllers. CMU's port-based objects [84] include a simple approach to data management for hard real-time applications, but the data management protocol supported by these port-based objects lacks flexibility. It also incurs substantial runtime overhead by periodically copying data between a global state table and local caches.

In contrast, MDARTS is a flexible object-oriented RTDBS architecture suitable for hard real-time applications. MDARTS consists primarily of a library of object-oriented database service classes. Tasks needing to share data with other tasks declare objects belonging to the MDARTS database classes. These objects are automatically registered with 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 object 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. A major difference between MDARTS and other RTDBSs is that local bus-based transactions on multiprocessors are accomplished by application tasks without communicating with a separate database server. We have implemented and tested a prototype of MDARTS and have demonstrated its utility in controlling an actual manufacturing machine.

Some unique features of MDARTS (compared to other RTDBSs) include:

- hard real-time guarantees,
- the ability to query a database object to determine its guaranteed transaction times prior to performing a transaction,

- customization of database services at object initialization time using application-side software contracts and an object-oriented technique called exemplar-based programming,
- transparent support for both remote procedure call transactions and local shared-memory transactions,
- multiprocessor support without performance bottlenecks or overhead associated with client-server architectures, and
- a convenient application programming interface.

In subsequent sections of this chapter, each of these features is discussed in detail.

An application using MDARTS declares database variables as in the following example:

```
MdartsArray<Point> positions("positions","read<=50usec;size=6;exclusive_update");
```

In this declaration, MdartsArray<Point> is the database class of the object positions, with which the application can store and retrieve elements from a shared array of Point data structures. Point is an application-defined data structure that contains a three-dimensional Cartesian coordinate. The two string parameters within the parentheses are passed to the MdartsArray<Point> constructor routines which initialize the positions object. 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). In our current prototype, this usually corresponds to local transaction execution time since the blocking times are zero. The following example shows how an application would perform a read transaction:

```
Point end_effector_position = positions[5];
```

Clearly, the syntax for using MDARTS is very convenient compared with application programming interfaces that require preprocessing of embedded query languages. Object-oriented database systems often have convenient application programming interfaces, so MDARTS is not unique in this respect.

Prior real-time database systems either make deadline guarantees *a priori* with off-line static analysis of applications [74] or use dynamic transaction scheduling to try to meet deadlines at runtime [12]. 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 object initialization. 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.

2.2 MDARTS Transactions

2.2.1 Transaction Properties

A transaction is a fundamental concept in database systems. According to Elmasri and Navathe, "The *execution of a program* that accesses or changes the contents of the database is called a **transaction**" [22]. Traditional properties of transactions (the so-called ACID properties) include atomicity, consistency preservation, isolation, and durability. Atomicity means the transaction is either completely performed or not at all. Consistency preservation means the transaction transforms the database from one consistent state to another. Isolation means the effect of a transaction is not visible to other transactions until it is committed. Durability means the effect of a committed transaction on the database is permanent. Elmasri and Navathe add serializability to the ACID transaction properties. Serializability is a concurrency control property that means the effect of running interleaved concurrent transactions is equivalent to running them in some serial order.

Some of the ACID properties of transactions can be expensive to provide, especially in the context of real-time systems. Durability can be particularly expensive since it usually implies keeping the database on a disk and incurring all of the overhead and execution time uncertainties associated with disk I/O. Atomicity and isolation can also be expensive, especially if a large fraction of the database is accessed or modified by the transaction. The serializability property has been the focus of recent debate in the real-time database community. Serializability is considered indispensable in the non-real-time database community, and some real-time database researchers concur [12, 55, 28, 29, 41]. Other real-time researchers consider full serializability too expensive and possibly unnecessary for real-time databases [48, 71].

Since many of the traditional ACID transaction properties may be expensive or infeasible to provide in the context of a real-time database, it may be desirable to limit or trade off various transaction properties. Unfortunately, the costs and benefits of various transaction properties depend on application semantics and database usage patterns. Some applications, such as real-time machine controllers, may not need persistence for data such as sensor values that are accessed within time-constrained feedback loops. Other applications may need persistence for some or all of their data. Similarly, the data consistency requirements of some applications may require serializable transactions while others may not. The difficulty with this situation is that no fixed set of transaction properties will be suitable for all applications. Therefore, MDARTS does not support a single set of transaction properties. Instead, MDARTS provides a framework within which multiple transaction types can be used. The transaction properties required by applications are explicitly declared and matched against the properties supported by various database service classes. In this way, an application can avoid incurring the overhead implicit in database properties that it does not need. Furthermore, the specification of properties is on a per-object basis, so the heterogeneity of data needs within an application can be closely matched by MDARTS services.

The internal heterogeneity of transaction properties in MDARTS makes it somewhat difficult to compare MDARTS with other real-time database approaches. MDARTS can support extremely fast transaction times for transactions that directly access non-persistent data in shared memory. Chapter 6 summarizes transaction timing experiments performed by our prototype MDARTS implementation. The worst-case transaction times in our experiments ranged from 25 to 180 microseconds, depending on the transaction. These transaction times correspond to worst-case object contention conditions, with multiple concurrent transactions performed using the same object. Since we use spinlocks that busy wait to enter critical sections, these transaction times are composed entirely of local task execution times rather than blocking times.

However, it is not fair to compare this absolute performance with that of a real-time database that uses file I/O for persistence. It would be possible to encapsulate access to a persistent database within an MDARTS object by invoking transactions on that database within the object's read and write methods. In that case, however, the speed of MDARTS transactions for that object would be constrained by that of the underlying database. MDARTS reflects the worst-case transaction times of its underlying data services, whether they are local memory transactions or transactions submitted to a file-based database system. Note that our prototype implementation does not provide interfaces to file-based database systems. However, we intend to create such interfaces in the future.

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 closing End_Transaction(). The application is allowed to perform an unlimited number of operations within the transaction, and the database is required to guarantee atomicity and other transaction properties for the entire sequence of operations. 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 make it more difficult to guarantee that lower-priority transactions will meet their deadlines. Furthermore, if the aborted transaction were close to committing, substantial processing resources could be wasted in forcing the transaction to start over. This issue is considered by Huang *et al.* in [35].

Unconstrained application-defined transactions may or may not be acceptable in the context of soft real-time systems such as considered in [35], but to provide hard real-time guarantees, a database system must tightly control transaction execution. Therefore, MDARTS transactions are modeled as object method invocations (function calls). An application can provide parameters to the transactions, and a transaction can perform relatively complex computations. However, MDARTS currently does not support application-defined transactions that span multiple database operations. We do not believe that such transactions can be supported in hard real-time systems without sacrificing serializability. In the future we may investigate providing limited support for more complex transactions, perhaps using data versioning [39], but we have not yet developed a model of such transactions or tried to implement one within MDARTS. With the current MDARTS transaction model, it is possible to guarantee transaction times while supporting atomicity, consistency, isolation, and serializability. The processing required to perform each transaction is specified entirely in the code of the database objects, so the database system can determine the transaction times either analytically or empirically. The only major source of execution time uncertainty in this context (assuming bounded latency to access system resources such as a shared bus) is that associated with concurrency control. With appropriate concurrency control protocols, MDARTS objects can bound this uncertainty and thereby guarantee their transaction times. The MDARTS transaction model also permits a simple distributed implementation using remote procedure calls. Locks in MDARTS are never held across the network since the transaction is propagated to a remote server and executed there as a local operation.

In summary, although transactions in MDARTS are more limited than traditional database transactions, we believe these limitations are necessary if the database is to make hard real-time guarantees.

2.2.2 Transaction Scheduling

Most of the algorithms proposed for real-time databases assume a client-server architecture in which a database process services multiple application tasks. When several transactions arrive at the same time, the server must choose the order in which to service them. This decision becomes complicated if timing constraints are associated with transactions and if transactions that access the same data must be serialized. In a real-time database, timing constraints are certainly associated with transactions. The question of whether transactions must be serialized has generated considerable interest in the RTDB community. However, the entire context of this debate has been framed by the architectures of traditional database systems.

Clearly, there are many important application domains where traditional file-based databases with enhanced real-time characteristics are needed. Program trading systems and on-line reservation systems are examples. Nevertheless, application domains such as manufacturing machine controllers have very different needs. The transaction-time requirements of systems like these are probably unachievable in the context of a traditional file-based database architecture. MDARTS uses the following methods to provide high-performance real-time data management:

- use memory rather than disks,
- use multiprocessors rather than uniprocessors, and
- avoid the overhead associated with a client-server architecture.

The motivation for the first two points is obvious: disk I/O times are relatively large and unpredictable, and multiprocessors can achieve extremely high performance through parallel processing. The third point is motivated by the need to achieve maximum parallelism and by the observation that as the basic cost of a service decreases, the overhead for providing that service becomes increasingly important. If tasks on multiple CPUs submit simultaneous requests to an object that resides on a single CPU, that object (rather the CPU and server task associated with it) becomes a serial bottleneck. Furthermore, if a server can perform an operation in ten microseconds but the overhead required to use the server is ten milliseconds, there is tremendous incentive to find alternatives that do not incur the overhead. These overhead numbers are realistic: a typical RPC round trip can take several milliseconds, whereas local procedure calls and memory accesses require only a few microseconds. Some high-speed RPC systems can reduce the overhead of transferring data and control between clients and servers [9, 46], but these systems are not widely available, they rely on processor-specific optimizations and low-level kernel modifications, and they do not eliminate the serial bottlenecks of server processes on multiprocessors.

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. This means that, with MDARTS, simple data read and write transactions can be performed at speeds approaching that of accessing ordinary variables. However, if the overhead of performing inter-process communication between clients and servers is added to each transaction, much of the speed advantage of the memory-based approach can be lost. MDARTS avoids the client-server overhead by allowing tasks to perform transactions themselves through their database objects.

By avoiding the client-server architecture, we recast the entire transaction scheduling problem. If there is no server managing multiple requests, where is the transaction scheduling? Put another way, how can we schedule concurrent transactions without a server? Each task performing a transaction must participate in the scheduling. MDARTS achieves this by embedding concurrency control and scheduling logic in its object methods. MDARTS objects are distributed across the tasks in a multiprocessor. The tasks themselves perform the execution of the database transactions. Therefore, none of the task switching and inter-process communication overhead implicit in conventional client-server architectures is incurred. Furthermore, the transactions are automatically performed at the respective priorities of the application tasks, and no system overhead is incurred to dynamically modify the priorities of server tasks. In our prototype implementation, we achieve serialization by simply serializing transaction critical sections with spinlock queues. Priority inversion is bounded by disabling preemption when a task is waiting for a lock or executing its critical section (Section 2.10 discusses this issue).

Since there is no blocking for I/O, and we have a parallel machine, serial access is a simple and efficient approach for mutually exclusive resource sharing. Note that only mutually exclusive transactions need be serialized. Concurrent transactions on different objects or compatible transactions on the same object can proceed in parallel with no synchronization delays. The MDARTS transaction model also helps make serial access acceptable by guaranteeing that locks are held only for very short periods of time. Transactions that must lock resources for significant time periods should use semaphore queues rather than spinlock queues so the processor is available for other tasks while the transaction waits for the resource (see Chapter 4).

The approach to scheduling transactions in MDARTS is completely different from that of other RTDBSs. By radically distributing transaction execution across the multiprocessor, we have transformed the problem from a local scheduling to a global scheduling problem. In general, global scheduling is more difficult than local scheduling. However, MDARTS does not really attempt to perform global scheduling of active transactions (in the sense of assigning transactions to available processors). Instead, 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. In many cases, a simple FIFO policy for an object's locking protocol will suffice. If the queue length can be bounded, a FIFO can provide transaction-time guarantees.

An object's concurrency control protocol in part determines its worst-case transaction time, and the transaction times are negotiated and verified during object initialization. Therefore, the acceptability of an object's locking protocol is indirectly verified at runtime during object initialization. Since MDARTS transactions proceed in parallel unless they conflict with other active transactions, MDARTS can take full advantage of the parallelism available on the multiprocessor. MDARTS transactions are not scheduled by any centralized scheduling task or processor, so MDARTS avoids serial bottlenecks on multiprocessors.

2.2.3 Nested Transactions

The current MDARTS implementation provides no direct support for nested transactions in the sense of guaranteeing serializability. However, it is possible for one object's transaction to invoke a transaction on another object. Since there is no global transaction management in MDARTS, it is the responsibility of the database object designer to ensure serializability if transactions invoke other transactions. The most direct way to achieve this is to create an interface class that implements application-specific transactions in terms of sub-transactions on other database objects. The interface class becomes the path through which applications perform relatively complex database operations on multiple objects. By encapsulating the sub-transactions on other objects, the interface object can simplify the application's interface and can enforce serializability of the nested transactions.

2.2.4 Real-Time Guarantees for Transactions

The most common approach to deadline guarantees in RTDBSs is to provide transaction processing that gives preference to higher-priority transactions. It is deemed the responsibility of the application or the operating system to assign appropriate priorities to tasks and thereby to the task's database transactions. In this model, the database server execution is seen as an extension of the execution of each task. Priority-based transaction processing protocols do not guarantee the transaction times. They only guarantee that higher-priority transactions will receive preferential service. Interestingly, in some cases, priority-insensitive optimistic concurrency control algorithms can outperform priority-cognizant transaction scheduling in the sense that fewer deadlines are missed as the transaction load increases [41].

Unfortunately, neither priority-based transaction processing nor optimistic concurrency control algorithms are particularly suitable for hard real-time applications. These protocols cannot prevent transaction overload conditions in which more transactions are submitted to the database than can be scheduled without missing deadlines. Furthermore, these protocols can abort and restart transactions in an unpredictable manner. It is difficult to analyze the computational resources required to service transactions that might be aborted and restarted. A database server using these techniques makes a best effort to satisfy the timing requirements of multiple transactions, but it makes no guarantee of individual transaction times. Therefore, it is possible that some application deadlines will be missed. In fact, the performance advantage of optimistic concurrency control algorithms over priority-based protocols derives in part from the database server discarding certain transactions that it determines cannot be completed before their deadlines. In a hard real-time application, missing even one deadline could be fatal, so this feature can hardly be considered an advantage.

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. It should also be possible to make these guarantees in advance, before the transaction is actually performed. In other words, the database system should require advance reservation of transaction resources to protect hard real-time applications from overload conditions. Prior RTDBS research has not considered the possibility of making transaction-time guarantees during initialization. From the perspective of a real-time task, the database transaction should be an atomic operation with a bounded, worst-case execution and blocking time. If the database provides this level of predictability, the task scheduler can guarantee higher-level task deadlines through a straightforward application of well-known real-time scheduling theory [50]. Guaranteeing each transaction's execution time also helps applications estimate the latency of tasks that perform database operations. This capability can be very important for systems in which the response time to events is critical.

Some real-time database researchers emphasize maintaining temporal consistency constraints in the data itself. MDARTS focuses on the performance of the database system and defers temporal semantics of the data to the application. If an application needs database transactions that preserve data temporal consistency, it is possible to implement new database classes with transaction methods that provide whatever semantics are needed. MDARTS is modular and easily extensible in ways like this.

Quantum mechanics notwithstanding, predictable systems are composed of predictable components. This principle motivated the MDARTS emphasis on providing predictable transaction times, and it guided our approach to implementing MDARTS itself. 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 plaform 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 transaction time guarantees) or be based on networking protocols and hardware that provide end-to-end response-time guarantees.

In a shared-memory multiprocessor, it is necessary to characterize delays associated with using the shared bus that provides access to the global memory. MDARTS transactions that use the multiprocessor bus will inevitably be delayed by the bus access latency. This is a factor over which MDARTS has no control, and it is highly implementation-dependent. Some 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, it is possible to determine worst-case latencies to access the bus. Given these worst-case latencies, MDARTS can guarantee its transaction times.

2.3 Object-Oriented Database Service Classes

2.3.1 Implementation Approach

We have implemented MDARTS in the C++ language [85]. We chose C++ because of its wide availability, runtime efficiency, compatibility with C, and object-oriented features.

Some advantages of C++ for real-time software are discussed in the literature [18, 17]. To enhance the portability of our implementation, we use only standard features of C++ rather than adding language extensions. It is beyond the scope of this dissertation to fully describe the C++ language. It is a very popular language with dozens of excellent reference books available. Therefore, we will only mention a few C++ features and terms.

C++ permits data structures and functions to be combined into new object types called "classes." Classes can inherit behavior (functions or methods, we use these terms interchangeably) and data from other classes. Classes that inherit from other classes are called *derived classes*, and those they inherit from are their *base classes.* C++ classes have special functions for creating and destroying instances of that class. These functions, called "constructors" and "destructors," are defined by the class programmer, and they are automatically invoked at runtime by the language. Constructors can be passed parameters which are used to initialize objects to some desired state. C++ also supports runtime polymorphism through virtual functions. This means a function defined as "virtual" in a base class can be overridden in derived classes, and the derived class version will be invoked at runtime even if the exact class of the object is not known to the caller of the function. This capability may seem strange and of dubious value to people unfamiliar with object-oriented programming, but it is very powerful and is one of the key features of C++. Finally, C++ allows class designers to redefine the meaning of standard operators such as "+", "*", and even function call operators "()" for objects of that class. This feature, called operator overloading, permits expressions involving objects of user-defined C++ classes to be as concise and convenient as ordinary arithmetic expressions.

MDARTS includes a base class (called Base) that defines the essential framework of MDARTS and contains various object initialization and data access methods. Base implements methods to construct new objects, communicate via RPC with the MDARTS Shared Data Manager, parse and check constraint lists, and store and retrieve the shared data. The RPC communication used by Base is performed during object initialization, before transaction-time guarantees are made. Base also specifies the interfaces of several methods that it does not implement. The MDARTS library requires that these methods be implemented by classes derived from Base (using the C++ pure virtual function mechanism). For example, the pure virtual function dataSize() returns the amount of shared memory required by the derived database service class. Because the Base function interface is inherited by derived classes, compatibility with the MDARTS framework is ensured. Furthermore, code reuse through inheritance reduces the amount of work required to integrate new database service classes into the MDARTS framework.

Figure 2.1 illustrates the structure of the MDARTS inheritance 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. This figure shows a single hierarchy corresponding to template array objects. In addition to array classes, MDARTS provides a generic database class hierarchy that defines transaction methods that set or get the basic data types of integer, floating point, and character strings. MDARTS also provides a template-based class hierarchy for application-defined data structures.

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

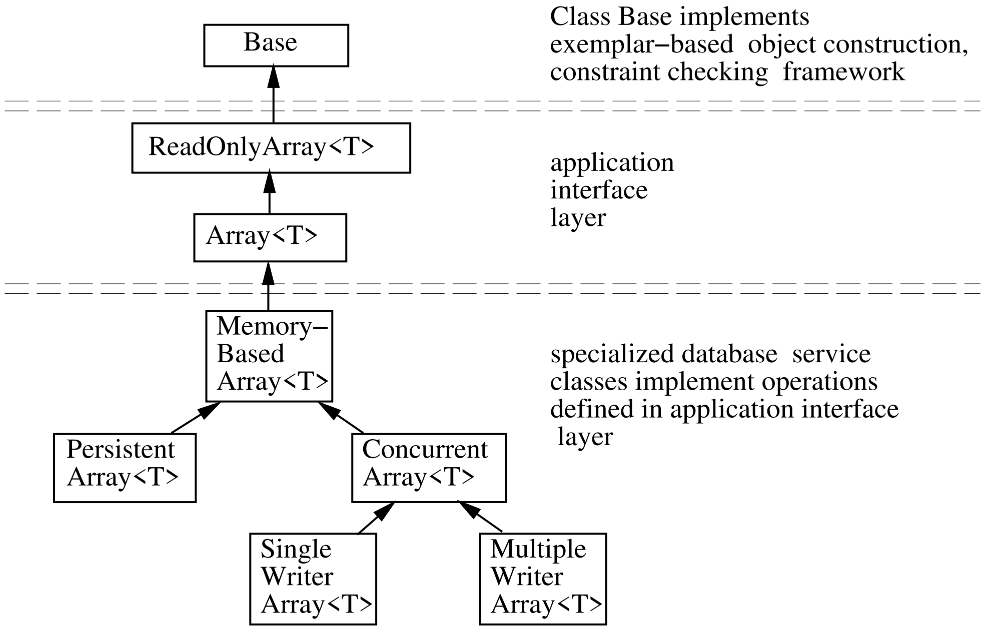


Figure 2.1: MDARTS database class hierarchy.

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. See Section 2.8 and Chapter 3 for further details relating to the construction of database service objects. 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 transactions is a form of delegation. With C++ inline functions and an optimizing compiler, very little runtime overhead is added through transaction forwarding. If even this small overhead becomes a problem, it is possible for an application to obtain a direct pointer to the database service object and use it instead of the interface object.

The database service classes in the MDARTS library function like populations of individual contractors. The contract analogy and its relationship to object construction is discussed in detail in Chapter 3. 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 [90, 91]. If multiple concurrent writers are allowed, additional overhead is required to lock and unlock the data and to wait if another task is updating it. The MDARTS library contains data management classes optimized for restricted concurrency semantics as well as classes that support more general semantics. Each database class guarantees transaction times according to its own implementation.

If none of the MDARTS classes can satisfy the contract, the application is notified (an exception is raised). Because the contracts are checked during object initialization, MDARTS detects possible transaction overload conditions before the transactions are actually performed. This point is crucial for safety-critical applications because it permits an application to verify its timing properties before beginning real-time processing. Once a data object has been constructed, the application program uses the object to perform database transactions. The location of the data and the implementation of the data access mechanisms are hidden from applications. Some objects directly access shared memory for maximum transaction speeds. Some objects use remote procedure calls to submit transaction requests to remote MDARTS SDM servers across the network. Other objects could access persistent data in file-based databases (we have not yet implemented persistent objects in MDARTS, but there are many techniques for accessing file-based databases using C++ classes [11]). In our current implementation, only shared-memory objects provide guaranteed transaction times.

The object-oriented architecture of MDARTS has been crucial to the success of the real-time constraint implementation effort. MDARTS is a relatively complex software system, but the programming interface presented to the user is remarkably simple. This is a direct result of the power and flexibility of the C++ language and the advanced object construction methods employed in MDARTS.

2.4 Memory-Based Objects

The timing constraints of some real-time applications are so tight that database transactions on the order of tens of microseconds are required. For example, an NGC manufacturing machine controller will typically have one or more tasks monitoring sensors and computing control signals. An NGC control task might be periodic with a hard deadline every millisecond. Each time the control task runs, it extracts the current sensor values from the database and computes new control signals for the machine actuators. Several database transactions may be required to compute the control signals each cycle. Therefore, the worst-case transaction times must be much less than one millisecond. Disk-based databases, or virtual memory-based databases that may generate page faults, cannot yet approach this speed (recall that we are talking about worst-case times). Therefore, data accessed at extremely high speeds must be kept in shared physical memory.

Real-time control systems typically keep their key data structures in memory as ordinary variables, but then the data is local to the control tasks and inaccessible to other software modules that might need to access them to perform execution monitoring or higher-level control. To permit more flexible data sharing, some systems make the memory addresses of

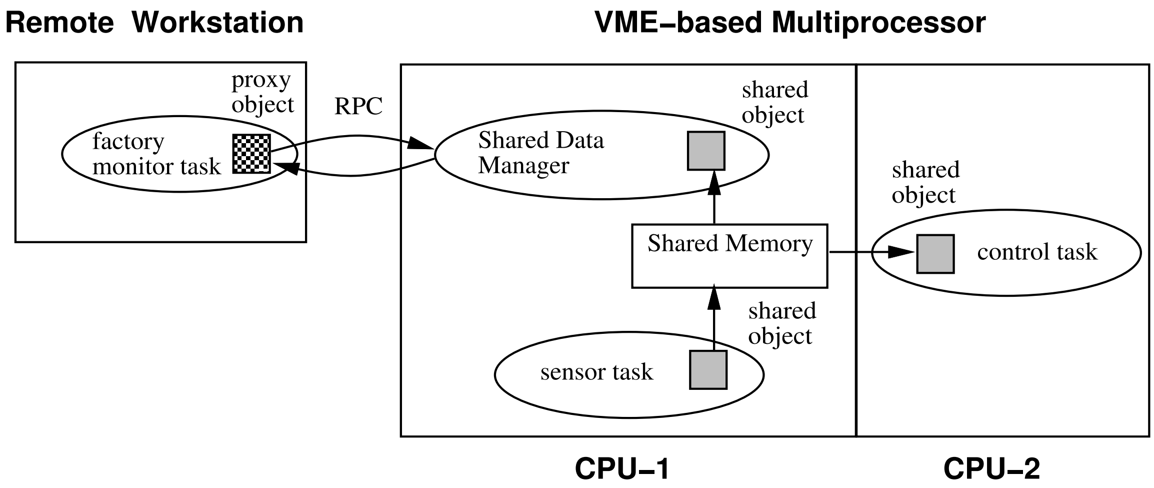


Figure 2.2: Access to shared memory data.

data objects known to multiple tasks by using pointers or declaring global data structures at predefined absolute memory addresses. With these types of sharing, there is a danger that some of the tasks will inadvertently misuse or misinterpret the data and possibly corrupt the common data areas. Such errors are extremely difficult to find and can have catastrophic consequences. In general, it is a bad idea for multiple tasks to directly manipulate pointers to common data areas. This is especially true in multiprocessor systems, where tasks can be distinct processes that run concurrently on different CPUs (i.e., they are not just different threads of a single process). A better approach is to encapsulate access to the common 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.

2.4.1 Design

Figure 2.2 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 2.2). 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. The execution of transactions by tasks also affects transaction scheduling. This issue is discussed in Section 2.2.2.

2.4.2 Implementation

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 [37] 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 object-oriented 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.

For example, part of an MDARTS class called "MdartsInt" is shown below. MdartsInt is a database service class. It contains a pointer (theInt) to a single integer in shared memory. The pointer is initialized by the virtual function setMemoryPtr(), which recurses up the class hierarchy to determine the shared memory address of theInt. SetMemoryPtr(), together with the virtual function dataSize(), permits future classes derived from MdartsInt to declare additional shared memory data structures. We will present the code and then describe this mechanism in more detail.

Once an object of type MdartsInt is initialized, its theInt pointer points to a valid shared memory address. This memory may be on the local machine, or it may be somewhere across a multiprocessor bus. Suppose an application task had an object of class MdartsInt named foo. This task could set the shared integer value of foo to 99 with the statement: foo.setValue(0,"",0,99); The shared integer value of foo could be retrieved with: int val = foo.getIValue(0,"",0); These methods are converted by the compiler to simple shared memory accesses which require at most a few microseconds to complete. The amount of time required depends upon the location of the shared memory and the performance of the computing platform. If the shared memory is on the same processor as the task and no virtual memory page faults are possible (the shared pages are locked into physical memory or virtual memory is simply disabled by the operating system), the memory read or write will

```
class MdartsInt: public RW_Mdarts {
typedef         RW_Mdarts inherited;
   int * theInt;         // a pointer to an integer in shared memory
   virtual int  getIValue(int itag, const char * tag, int index)  {
      DECLARE_BUS_OPS(1)
      return *theInt; }
   virtual void setValue(int itag, const char * tag, int index, const int val)  {
      DECLARE_BUS_OPS(1)
      *theInt = val; }
   virtual size_t dataSize() { return inherited::dataSize() + align(sizeof(int)); }
   virtual void * setMemoryPtr(void *ptr) {
      ptr = inherited::setMemoryPtr(ptr);
      theInt = (int*) ptr;
      return ptr + align(sizeof(int));
      }
   // ... other MdartsInt methods omitted
};
```

Figure 2.3: Code for MDARTS class MdartsInt.

require only a fraction of a microsecond. If the shared memory is across a multiprocessor bus, the memory access will be delayed by the time required to gain control of the bus and perform the bus transaction. This delay in turn depends upon the availability of the bus when the request is made. A properly-designed real-time platform should ensure that enough bus bandwidth is available to guarantee access to the bus within a few bus cycles. In this case, the read or write should take no more than a few microseconds.

The "itag" and "tag" parameters to getIValue() and setValue() are used to identify which members of a complex MDARTS object should be returned. For example, suppose an MDARTS class stored a structure with six integers in shared memory rather than a single integer. The task calling getIValue() could indicate which member it wanted by passing a code for that member in "itag" or "tag." In our simple MdartsInt example, these parameters are ignored. However, it is necessary to include them in the function declarations so that they match the virtual functions declared by the MDARTS base classes. The "index" parameter is used for array objects or stream objects for which a cursor (or iterator) is needed. We discuss these issues further and explain how the syntax for performing transactions is simplified even more in Section 2.9.

The careful reader will notice that the name of the read function, getIValue(), has encoded in it the data type to be returned (an integer), while setValue() does not. C++ permits defining multiple functions with the same name, provided their parameter types differ (this is called function overloading). However, the getIValue() functions all have the same parameters, so we are forced to use different names. Multiple methods named setValue() can be disambiguated by the compiler by virtue of the different types of their fourth parameter. The getValue() methods do not have a fourth parameter, so it is necessary for them to have unique names. As an alternative, an overloaded getValue() function could be defined if it took a fourth parameter to store type-specific function results. We do not consider this issue particularly important, especially since the methods described in Section 2.9 hide this complexity from the application programmer.

Having introduced the methods that perform transactions, we will explain the process

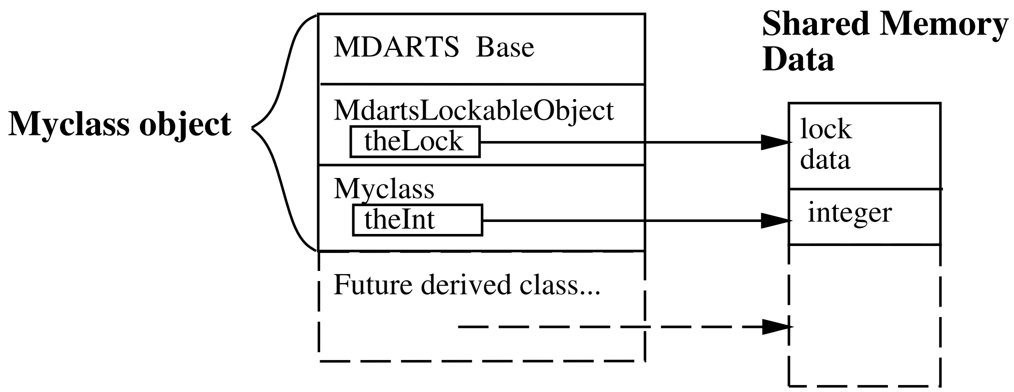


Figure 2.4: MDARTS pointers to shared memory regions.

of initializing the shared memory structures. When the Shared Data Manager is ready to instantiate a new instance of a MdartsInt object, it calls the MdartsInt function dataSize(). This function recurses up the inheritance hierarchy, returning the overall amount of shared memory needed for a MdartsInt object. Note that this size is not the same as the value returned by the standard C++ function sizeof(MdartsInt). Sizeof(MdartsInt) returns the size of the local part of a MdartsInt object, not the part in shared memory. The shared memory size of any class X is recursively defined as the shared data size of its base class plus the aligned size of the shared data of X. The call to align() insures that derived classes will be able to put arbitrary data at the address just beyond the end of the shared memory used by X (some machine architectures require long words, pointers, or floating point numbers to be placed at long word boundaries in memory). When creating an MdartsInt object, the SDM allocates MdartsInt::dataSize() bytes of shared memory, say at address addr, and calls MdartsInt::setMemoryPtr(addr). Like dataSize(), setMemoryPtr() recurses up the class hierarchy to determine which shared memory address theInt should contain. In the recursion, the shared memory pointers of any base classes of MdartsInt are similarly initialized. This mechanism permits new classes to be derived from MdartsInt that include additional data to be placed in shared memory.

Figure 2.4 illustrates the data parts of a shared MDARTS object of type MdartsInt once the shared memory pointers have been initialized. The dashed lines represent the potential for future classes derived from MdartsInt to add their data to the shared memory region without recompiling the code for MdartsInt and RW_Mdarts.

Although our approach to shared C++ objects works and is portable, it has some limitations. In particular, the data kept in shared memory is not permitted to contain pointers to other MDARTS objects (it can, however, contain pointers to other shared memory data structures). This makes it inconvenient to place MDARTS objects in shared data structures such as linked lists and trees. It is possible to devise various techniques to circumvent some of these limitations, but the result will not be as efficient or as convenient as structures composed of ordinary C++ objects. For example, shared linked lists can be created within MDARTS if they consist of lists of data structures (or objects without virtual functions) rather than lists of MDARTS objects. However, suppose we need a shared linked list of MDARTS objects. It is possible for each object to contain a pointer to its shared memory data and to the next object in the list (see Figure 2.5). However, this is not truly a shared linked list, because if one of the tasks performs a list insertion or deletion the list in the other task remains unchanged. MDARTS does not dictate the implementation of the shared

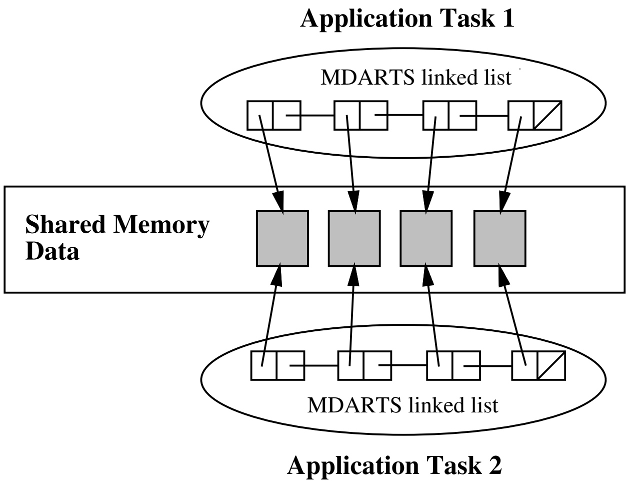


Figure 2.5: An MDARTS linked list.

objects, so these problems are caused by the implementation approach we have taken. If portability is not a requirement and virtual memory is available, Jordan's methods can be used for placing objects with virtual functions in shared memory.

2.5 Remote Transactions

In some cases, it is impossible to directly access the memory where data reside because the processor on which the task is running does not have physical access to it (i.e., the processor is not on the same bus). If distributed shared memory is available, it is possible to use physically remote memory as if it were local. However, most systems require some form of networking to access remote data. One of the difficulties associated with remote data access is that most networking protocols add substantial overhead and do not provide end-to-end response-time guarantees. For example, TCP/IP-based protocols for socket and datagram communications, on which Sun RPC [10] and OSF DCE [69] are built, provide no timing guarantees. Clearly, any remote transaction that uses these services cannot provide any absolute timing guarantees. Nevertheless, it can still be very useful to provide remote access even if transaction-time guarantees are not made. For example, suppose a factory monitoring task needs to examine the internal state of a particular machine controller. The factory monitoring task may not have tight deadlines, so it might be able to tolerate the latency and unpredictability of a transaction that uses RPC. Since markets for real-time networking to support digital multimedia and telecommunications are expanding rapidly, implementations of networking protocols with guaranteed response times should soon become widely available. Thus, soon it should be easier to provide remote data services that are suitable for real-time systems.

2.5.1 Design

One of the design goals of MDARTS is that data location and implementation be transparent to applications. Applications specify the database identifiers of the objects they need, and the MDARTS library constructs the appropriate objects to access that data. For local data (on the same processor or accessible across the bus), MDARTS objects with direct pointers to shared memory are constructed. For remote access, proxy MDARTS objects are constructed. Proxy objects contain the information needed to perform RPC transactions on that object. This information includes the object identifier and the RPC handle corresponding to the SDM server that manages that object. When making transaction-time guarantees, MDARTS takes communication delays into account. In our current implementation, this essentially means that if MDARTS determines remote access is required, it will refuse to guarantee transaction times for the proxy object. The location of data is not completely transparent to applications since a timing constraint that is supported for local data will be rejected if the data is remote. This behavior is useful since it prevents unsuspecting applications from performing operations that might cause them to miss their deadlines. If we were using networking protocols that provided end-to-end response time guarantees, MDARTS could be enhanced to also guarantee transaction times for proxy objects [38]. However, this would still not be entirely trivial since the scheduling of RPC requests in the SDM would have to be considered. To do this, the SDM would need to have sufficient execution time to service remote requests. The CPU utilization required by the SDM would depend on the arrival rate of remote requests and their worst-case execution times.

For remote access to data, our implementation of MDARTS uses remote procedure calls (RPC) [10]. Applications that need to access remote objects use proxy objects that generate RPCs to ask the SDM to perform transactions on their behalf. Applications need not know whether the access is local or remote; the MDARTS objects hide the access mechanism. In other words, when applications construct their MDARTS objects, they do not specify whether the objects are local or remote. The MDARTS object construction process determines what type of object is required for each task to access each database object.

Naturally, remote transactions that use RPC are much slower than transactions that use direct shared memory access. For example, on our test platform a local database transaction that requires no more than 20 microseconds for direct memory access typically requires 30 milliseconds if a proxy object is used. Almost all of this time is consumed in the RPC overhead required to send messages across the Ethernet. Faster processors, such as Sun Sparcstations, can perform RPCs on average significantly faster than this. However, since the TCP/IP protocols do not guarantee delivery times, the higher average performance does not help with respect to hard real-time guarantees.

It is crucial that the execution of the RPC-based transactions not cause local real-time tasks that use the same data to miss their deadlines. For this reason, our initial transaction model does not support compound transactions that allow applications to lock resources across multiple data access operations. If such transactions were allowed, locks held across the network by remote applications could block execution of local transactions for extended periods of time, making it impossible to guarantee fast transaction times for the local tasks. Instead, in MDARTS, each transaction is essentially equivalent to a single local or remote procedure call. If a transaction requires a lock, it obtains and releases the lock before it returns control to the application. In the case of a remote transaction, the lock is acquired by the Shared Data Manager, which services the RPC call, performs the transaction specified in the call, and returns the result to the application through the remote proxy object (see Figure 2.2).

```
1 – class MdartsRemote: public MdartsLetterBase {
2 – public:
3 –     virtual int  getIValue(int itag, const char *tag, int index) {
4 –         int answer;
5 –         get_ival(theHandle, theObjectName, itag, tag, index, &answer);
6 –         return answer;
7 –         }
8 –     virtual void  setValue(int itag, const char *tag, int index, const int val) {
9 –         set_ival(theHandle, theObjectName, itag, tag, index, val);
10 –        }
11 –
12 – protected:
13 –     CLIENT *   theHandle;       // RPC handle
14 –     char *          theObjectName;  // object identifier
15 –
16 –     // ... other MdartsRemote methods omitted
17 – };
```

Figure 2.6: MDARTS proxy class for client side of RPC.

2.5.2 Implementation

To implement MDARTS remote access, we created a proxy class called MdartsRemote. This class uses RPC calls to implement the virtual functions that store and retrieve data. The RPC calls are serviced by MDARTS SDM servers. Each MdartsRemote object contains an RPC handle and the identifier (name) of the remote MDARTS object. Our first MDARTS implementation used Apollo NCS for its remote procedure calls. NCS is the base technology for OSF DCE remote procedure calls [69]. However, since NCS and DCE are not widely available yet, we modified the RPC functions in MDARTS to use Sun RPC [10]. Figure 2.6 illustrates some of the MdartsRemote class members that forward transactions via RPC. The functions get_ival() and set_ival() (called on lines 5 and 9, respectively) bundle their arguments into the structures used for the Sun RPC client stubs and generate the RPC calls.

Figure 2.6 represents the client side of the RPC call. The server side is shown in Figure 2.7. Get_ivalue_1() is the remote procedure provided by the MDARTS Shared Data Manager to service transactions that retrieve integers from MDARTS objects. The object is retrieved by name from a hash table of registered objects (line 5), and then the getIValue() function is called for that object. In this way, the SDM performs the transaction on behalf of the remote proxy object and returns the results via RPC.

The call to getIValue() on line 12 of Figure 2.6 illustrates the power of virtual functions in C++. Notice that get_ivalue_1() never tries to determine the actual type of the MDARTS object retrieved from the hash table. It simply calls the virtual function getIValue(), and C++ automatically ensures that the correct version of getIValue() will be called for that object. Thus, the RPC server code never needs to be recompiled when new MDARTS classes are created. The code for the new classes is linked into the SDM server, and no modifications of existing code are required. Therefore, MDARTS is very easy to extend with application-defined classes.

```
get_Irecord * get_ivalue_1(get_value_args *args)
{
    static get_Irecord *get_rec = new get_Irecord;

    Base *bp = getObjectPtr(args→name);        // look up the object by name
    if (!bp) {
        get_rec→st = OBJECT_NOT_FOUND;
        return get_rec;
    }

    // eventually put exception handling around the getIValue call.
    get_rec→answer = bp→getIValue(args→itag,args→tag,args→index);
    get_rec→st = RPC_OKAY;

    return get_rec;
}
```

Figure 2.7: GetIValue() Function for server side of RPC.

2.6 Real-Time and Semantic Constraints

Transaction deadline specification in real-time databases is important because real-time applications often contain implicit assumptions about the performance characteristics of data access operations. For example, the control loop of a machine controller might access a database to read sensor values or issue actuator commands. The sampling frequency of the control loop is determined *a priori* according to the control strategy and the physical characteristics of the machine. The sampling frequency helps determine the control task's deadline. To verify that the deadline will be met, the software designer must know the time required to complete the database transactions. However, if this timing dependency is implicit and is not checked at runtime, the application may fail catastrophically if the designer's assumptions about the database performance are wrong. Therefore, applications should specify their real-time constraints explicitly and delegate the responsibility for meeting them to the database system.

Prior real-time database research investigates algorithms suitable for meeting real-time requirements. However, relatively little attention has been paid to the question of how real-time applications should communicate their requirements to the database system. This issue is very important in any practical implementation. A common assumption is that transactions will contain deadline information and/or task priorities. The problem with putting deadline requirements in transactions is that if the database system cannot meet a deadline, this will not be discovered until the transaction is performed. Furthermore, processing deadline information adds additional overhead and complexity to transaction processing. This overhead in turn reduces the performance of the database system. DiPippo and Wolfe [21] present an object-based RTDB model in which object methods are specified with worst-case execution times and temporal scopes defining the timing behavior of subsequences of statements within the method. It appears from their paper that the database class designer is responsible for specifying all of this information, but the authors do not provide any tools to help perform the specification. In MDARTS, timing knowledge is derived empirically with object methods that benchmark transactions on the actual execution platform (see Section 2.7). This timing knowledge is used at object initialization time to evaluate application-specified transaction deadlines expressed as transaction-time requirements. By checking requirements during initialization rather than during transac-

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

Figure 2.8: Examples of real-time and semantic constraints.

tion execution, problems are detected early, and overhead during transaction processing is reduced.

Recent work by Badrinath and Ramamritham [7] and DiPippo and Wolfe [21] propose concurrency control techniques that use the semantics of object methods to increase the level of concurrency supported by database objects. The semantic information used by these techniques is limited to the state of the object and knowledge about the compatibility of the object's methods. Semantics-based concurrency control is a powerful technique to reduce the amount of locking in transactions. However, it is possible to improve transaction performance even more if application-level semantic information relating to the intended use of the object is available to the database. For example, suppose only one task in a distributed application should be permitted to update a particular object. In that case, concurrency control protocols that are tuned to single-writer semantics could be used. However, a database system ordinarily would not know that there will be only one task performing updates to an object. Therefore, the database is forced to use less efficient protocols that assume more pessimistic application semantics. MDARTS allows applications to declare semantic information at runtime to help match database services with application characteristics. This semantic information, along with transaction-time requirements, is passed to the MDARTS library object construction methods. Therefore, MDARTS is capable of using both object-level and application-level semantic information to customize services and maximize transaction concurrency.

2.6.1 Design

Figure 2.8 shows some example MDARTS constraints and how an application can specify them in contracts using character strings. An MDARTS contract is a string containing a list of constraints separated by semicolons. "Staleness" specifies an external consistency constraint. If a sensor monitor fails and its database values become obsolete, a staleness constraint can trigger an exception or warning when a task tries to read that data.

MDARTS provides a rich environment for developing database constraints. For example, researchers interested in new concurrency control techniques can implement their algorithms in database service classes within the MDARTS framework by deriving new classes from existing MDARTS classes. Since constraints are expressed as character strings, it is possible to invent a new semantic constraint, define a syntax for the constraint, and integrate it seamlessly with MDARTS. It is not even necessary to recompile the MDARTS library to add a new service class or a new constraint. New user-defined MDARTS classes that support the new constraints can simply be linked into the application code along with the standard MDARTS library.

```
1  – class Base {
2  –     enum ConstraintType
3  –       { unknown, null, size, type, restrict_type, read, write,
4  –         remote_access, memory_access, priority, range_checked,
5  –         access, staleness, persistence, concurrency, units };
6  –     // ...
7  – };
8  –
9  – class Constraint {
10 – public:
11 –     Constraint() : ct(Base::null), con(0), op(0), arg(0) { }
12 –     Constraint(char * c, char * o, char * a);
13 –     ~Constraint();
14 –     Base::ConstraintType        ct;
15 –     char * con;
16 –     char * op;
17 –     char * arg;
18 – };
```

Figure 2.9: Structures for constraints.

Application-specified constraints can be used for a variety of purposes in MDARTS. The constraint strings that are included in a request to create a new object are used to select the database service class and to configure the new object's state. Constraint strings that are passed in subsequent requests to share an existing object are used to verify that the new request is compatible with the capabilities of the existing object and will not cause any prior timing guarantees to be violated. One rather nice feature of the MDARTS contracts is that mnemonic tags can be associated with constraints that initialize various members in the objects. For example, an MDARTS array object might be initialized with the following statement:

MdartsArray<Point> sensors("sensors","read<=20usec;size=50;exclusive_update");

Notice that the size of the array is specified in the contract string. Configuration parameters like this are ordinarily passed to object constructors, but usually they must be passed in "raw" form, without the mnemonic "size=" tag to indicate their meaning. In simple classes, there may not be confusion regarding the meaning of the constructor parameters. However, if there are several configuration parameters passed to the constructor, it is possible to confuse their meaning. For example, suppose an ordinary C++ object corresponding to an array of integers has a constructor that takes two integer parameters. The first parameter establishes the size of the array, and the second one is used to initialize the elements of the array to that value. The declaration would look like:

IntArray sensors(10,20); // *is the size 10 and initial value 20 or vice versa?*

2.6.2 Implementation

When the MDARTS Base constructor processes a contract, it parses the string and converts it to a linked list of Constraint objects. The declaration of the ConstraintType enum on lines 2–5 of Figure 2.9 lists a basic set of constraint types defined for the MDARTS library. This list represents a preliminary attempt to define some constraints that might be useful for a real-time database. New constraint types should be added to this list as they prove useful for general-purpose database service classes.

When an application-specified contract is parsed by MDARTS, the string is subdivided

```
1  – class MdartsInt : public RW_Mdarts {
2  –     virtual int checkConstraint(const Constraint& c) {
3  –             switch  (c.ct) {
4  –             case size:
5  –                 if (SAMESTR(c.op,"=")) {
6  –                     shared→size = atoi(c.arg);
7  –                 }
8  –                 return 1;
9  –             // ...
10 –             default: return inherited::checkConstraint(c);
11 –             }
12 –     }
13 –     // ...
14 – };
```

Figure 2.10: Code for constraint checking.

into individual constraint clauses separated by semicolons. Each constraint clause is further parsed to extract a constraint type field and optional operator and argument fields. For example, the constraint "size = 10;" has constraint type "size", operator "=", and argument "10". When the constraint type is parsed, the string is used to choose a corresponding enumerated type for that constraint. This constraint value is used to initialize the "ct" member of a Constraint object (see line 14 of Figure 2.9). The string values of all three parts of the constraint clause are also stored in the Constraint object. Note that the very first constraint type on line 3 is "unknown". This type is chosen if the string for the constraint type does not match any of the predefined constraints. In this case, the actual string of the constraint type will have to be examined by the MDARTS class to determine if it is supported. Since MDARTS allows unknown constraints, it is possible to extend the number of constraints without modifying the existing MDARTS library. This is a very important feature.

Figure 2.10 shows part of the constraint checking method from a sample MDARTS class. Note that this method checks only a single constraint at a time. The MDARTS Base class provides a method that iterates over the list of constraints corresponding to an MDARTS contract and submits each one to the checkConstraint method. CheckConstraint() is a virtual function, so the version implemented in the most derived class will be called. A return value of 1 indicates success, and 0 indicates failure. On line 10, checkConstraint() delegates the checking of unrecognized constraints to the checkConstraint() methods of its base class. This is a very important point, because it allows a derived class to only check those constraints that it implements differently than its base classes. This significantly reduces the effort required to create a checkConstraint() function for a new MDARTS class.

MDARTS provides a framework and a mechanism for expressing and checking application-specified constraints. However, the validity of each constraint check is deemed the responsibility of the database class that performs the check. There is no protection against database classes that erroneously agree to a constraint that they actually cannot meet. Therefore, the real-time guarantees provided by MDARTS are only as good as the implementation of the classes that make the guarantees.

2.7 Benchmarking Execution Times

We have examined 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 [36]. 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 [36] 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. In other words, "on what CPU, what bus, and what clock rate does real-time method `m33` require 30 milliseconds to execute?" What if one wanted to compile the same source code to run on a multiprocessor with a mixture of 68030 and 68040 CPU boards? 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.

Since computers execute instructions at a well-defined rate and higher-level functions are composed of sequences of instructions, one might think that it would be easy to determine the execution time of a function by analyzing the source code. Unfortunately, this is a variant of the famous halting problem, which is theoretically undecidable. If restrictions are placed on the code, such as prohibiting looping constructs, it becomes theoretically possible to synthesize execution times. Because of the transformations performed by the compiler during code generation, one must analyze the assembly code rather than high-level source code. However, with modern CPU architectures that employ instruction and data caches, even analyzing the execution time of assembly can be difficult. Instead, 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, 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.

2.7.1 Design

Transaction-time guarantees in MDARTS are derived from benchmarking of method execution times, runtime estimation of locking delays, and estimation of worst-case bus access times. Execution time benchmarking can be performed in a separate calibration run of the MDARTS software on the target platform, or it could be performed at application initialization time. An MDARTS object includes a virtual function called `calibrate()` that performs a set of timing experiments on its methods. The results of these timing experiments can either be kept in memory or be output in a form that can be included in the class source code, which is then recompiled. The former case is used for benchmarking at initialization time. The latter case is used for a separate calibration run. Since MDARTS adds code to the locking methods to collect some of the timing information, overhead during transaction execution can be reduced by performing a separate calibration run and recompiling without the timing support. Inclusion of the timing analysis code is controlled by preprocessor directives, so it can be removed when the program is recompiled. Furthermore, the overhead of performing benchmarks during initialization can be avoided if the timing information is collected in a calibration run. Therefore, in most cases we prefer to use calibration runs to generate benchmarking information.

The advantage of lower overhead with calibration runs must be weighed against the possibility that the runtime platform differs in some significant way from that used in the calibration run. 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 correct benchmark can be included via preprocessing directives or could even be determined at runtime using a flag variable to indicate the CPU type. 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 standard function units. Suppose this code is then run on a 68040 on which the standard function requires only 10 microseconds. 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 method for determining worst-case wait times for locks is described in Section 2.10.

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. For example, some queryable fields might require the database object to perform a computation to derive the requested value while others might simply retrieve a value from memory. 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.

For example, an array object might support one read transaction that returns its size and one that returns the sum of the elements in the array. If the array is large, there could be a significant difference in the execution times of these two transactions. MDARTS permits each transaction method to have as many parameter-specific timing records as the class

```
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 2.11: Example of MDARTS benchmark data.

implementer deems worthwhile. The cost of providing more detailed timing information is nominal: less than 100 bytes per timing record for each class (all objects of a given class share the same timing records). The time spent searching the extra timing records depends on the search algorithm. With binary search, a record can be found in $O(\log n)$ time. Furthermore, most timing checks are performed during object initialization, and the time required to do the check does not affect the real-time guarantees of other transactions. In summary, the MDARTS benchmarking approach allows the database class developer to support timing constraints of very fine granularity an nominal cost.

2.7.2 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 ouput by a calibration run, this information is encoded into an array of character strings associated with that database class. Figure 2.11 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 that returns the sum of the elements in an array or linked list iterates over all of the elements. The number of bus operations for this transaction will be proportionate to the size of the array or the length of the list. Although our current implementation only supports constants and a linear modifier "x", it would be very straightforward to add other scale modifiers (e.g., "logx" for transactions that perform searches on binary trees). 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 2.12 shows the implementation of a calibration function corresponding to Figure 2.11. 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

```
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 2.12: Example of MDARTS calibrate function.

standard 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. Note also that the timing value for "write(start_motors)" includes a mixture of time values. Execution times for certain operations might include both computation times and delay times associated with waiting for events in the real world. Not every operation will scale directly with CPU performance. Therefore, it can be desirable to include an absolute time value plus some execution time scaled to CPU speed. Benchmarks such as this must be generated by hand, since the timing method cannot distinguish fixed overhead from CPU-dependent overhead.

```
#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 opv = 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/opv,units,\
    busOperations(),scaleFactor(),\
    (double) Lock::CsectionTime()/opv,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 2.13: Macros used in MDARTS calibration functions.

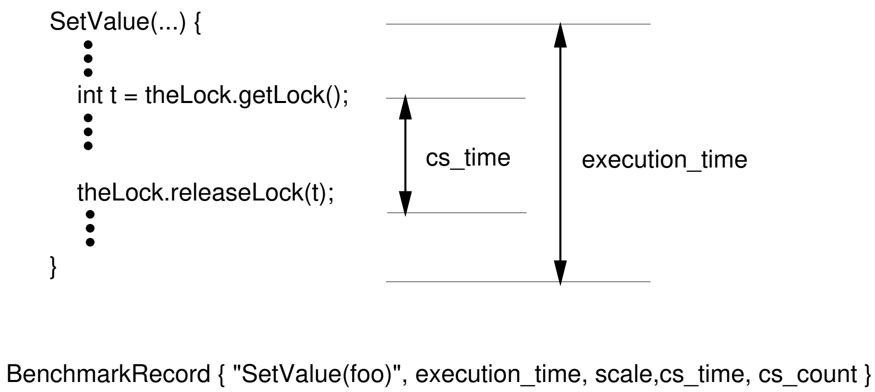


Figure 2.14: MDARTS method timing.

The macros used in Figure 2.12 are defined in Figure 2.13. These macros are the ones used to generate output during a calibration run for an MDARTS class. Similar macros without the print statements can be used for initialization-time benchmarking. 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. A Time value with the highest bit = 0 represents a time interval from 0 to 21 seconds with a resolution of 10 nanoseconds. If the highest bit is set, the lower bits are interpreted as seconds (with a 21-second offset) rather than 10-nanosecond units. By encoding time with this dual-resolution technique, intervals from 10 nanoseconds to 68 years can be represented in a single 32-bit value.

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. These macros are implemented in terms of hardware timers on the target CPU platform. On the 68030 boards of our multiprocessor, these macros use a hardware timer with resolution of 6.25 microseconds. On the Sun Sparc, we use an internal timer with 1 microsecond resolution. However, the interface to the timer on the Sparc is through system calls that introduce jitter and uncertainty of a few microseconds in the measurement. Figure 2.14 illustrates the execution time intervals measured by the RUN macro.

One of the objectives of the MDARTS benchmarking design was to minimize the programming effort and runtime overhead required to support timing analysis. Most of the timing code is localized in the calibrate() method. However, the MDARTS transaction methods for each object must indicate the number of bus operations and the size scale factor needed by calibrate(). To minimize runtime overhead, the benchmarking macros are conditionally compiled only for calibration runs (see Figure 2.15). After calibration, the preprocessor substitutes the null versions of the macros to eliminate the overhead they represent. Figure 2.15 shows the getDValue() method delegating to getIValue(). The DECLARE_BUS_OPS() macro increments a counter so that methods that call other methods need not contain knowledge of how many bus operations are invoked by other methods.

```
#ifdef CALIBRATE
#define DECLARE_BUS_OPS(busops) theBusOperations += (busops);
#define DECLARE_SCALE(scale) theExecutionScale = scale;
#else
#define DECLARE_BUS_OPS(busops)
#define DECLARE_SCALE(scale)
#endif
class MdartsInt: public RW_Mdarts {
   int * theInt;        // a pointer to an integer in shared memory
   virtual int  getIValue(int itag, const char * tag, int index)  {
      DECLARE_BUS_OPS(1)
      return *theInt;
      }
   virtual double getDValue(int itag, const char * tag, int index)  {
      return (double) getIValue(itag,tag,index);
      }
   // ...
};
```

Figure 2.15: Support for benchmark calibration in transaction methods.

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". 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. This permits applications to perform more sophisticated timing analysis than can be conveniently accomplished with constraint strings. Contract strings permit timing information to flow in one direction: from the application to the database object. With QueryTiming(), MDARTS supports the flow of timing information in both directions. Thus, an application can specify its minimum requirements of the database in the contract string and then query the resulting object for its actual performance, which may exceed the initial specification.

2.8 Construction of MDARTS Objects

Many of the unique capabilities of MDARTS are associated with the way MDARTS objects are constructed at runtime. There is no single answer to the question "how are MDARTS objects constructed?" The answer depends on the context in which an MDARTS object is declared and on the prior state of the database. To clarify some of these issues, we will describe the various ways objects are created and discuss the implications of these alternatives.

In 2.3.1, we discussed the MDARTS class hierarchy and mentioned that applications are shielded from knowing the exact class of the database objects they use. Applications interact with the database through classes defined in the application interface layer. C++ inheritance is used to enforce interface compatibility among derived database service classes.

Suppose an MDARTS interface class RW_Mdarts is the base class of several database service subclasses $Service_i$. If an application wants to use an MDARTS object of class RW_Mdarts, it can declare an object as: RW_Mdarts myob("myob", "read<=1msec"). This declaration automatically triggers RPC requests to the various MDARTS SDM servers to find an existing shared object called "myob." The MDARTS SDM that manages "myob" verifies that the existing object is compatible with the constraint "read<=1msec" and that the class of "myob" is compatible with RW_Mdarts (i.e., it is one of the classes $Service_i$). If a new object is to be constructed, an additional CREATE flag is supplied to the RW_Mdarts constructor. The SDM contacted with the create request verifies that the object does not exist already and then selects an appropriate service class $Service_j$ that can meet the specified constraints. Once the service class is chosen, the SDM creates an instance of that class and allocates whatever shared memory is needed for it. After the existing object is found or the new one is constructed, the SDM returns the type $Service_j$ and the shared memory pointer to the client application, which can then finish constructing its local instance of the shared object.

A task might want to use an MDARTS class for its local purposes without registering it with the SDM or making it visible to other tasks. The constraint-checking features of MDARTS can be useful even for objects that are not shared. In this case, it is unnecessary to assign a database name to the object or to communicate with the SDM. A task can create a purely local MDARTS object by simply omitting the database name in the object declaration. Each MDARTS class has a constructor that does not have a database name argument. This constructor does not issue any RPC calls but instead builds an object that meets the constraints in local memory. It is this constructor that the SDM uses when it services object construction requests. Therefore, there are three different MDARTS object construction sequences, depending on the context of the declaration. The first sequence corresponds to creating a new shared object; the second corresponds to creating a local instance of an existing shared object; the third corresponds to creating a new local object that is not visible to other tasks.

If a new shared MDARTS object is created with this declaration: RW_Mdarts ob("ob","read<=1msec",CREATE), the following activity occurs.

1. RW_Mdarts() constructor sends its parameters via RPC to an MDARTS SDM
2. the SDM searches those database service classes that are compatible with RW_Mdarts and constructs an object from one that is compatible with the timing constraint.
3. the SDM returns the exact database service class of the new "ob" and the location of its shared memory. If the remote client does not have access to the shared memory, a

flag is set to create a remote access object if timing constraints can still be met.

4. RW_Mdarts() constructor builds a local copy of “ob” with the same database service class (MdartsInt) and initializes its shared-memory pointers.
5. RW_Mdarts() constructor keeps a pointer to the local “ob” for subsequent transaction delegation.
6. RW_Mdarts() constructor returns.

For a local instance of an existing shared object, this declaration is used: RW_Mdarts ob("ob", "read<=1msec").

1. RW_Mdarts() constructor sends its parameters via RPC to an MDARTS SDM
2. the SDM determines if the existing “ob” is compatible with RW_Mdarts and if it can meet the new constraints.
3. the SDM returns exact database service class of the existing “ob” and the location of its shared memory. If the remote client does not have access to the shared memory, a flag is set to create a remote access object if timing constraints can still be met.
4. RW_Mdarts() constructor builds a local copy of “ob” with the same database service class (MdartsInt) and initializes its shared-memory pointers.
5. RW_Mdarts() constructor keeps a pointer to the local “ob” for subsequent transaction delegation.
6. RW_Mdarts() constructor returns.

For a purely local object, this declaration is used (note that no database name is included):
RW_Mdarts ob("read<=1msec").

1. RW_Mdarts() constructor searches those database service classes that are compatible with RW_Mdarts and constructs an object from one that is compatible with the timing constraint.
2. RW_Mdarts() constructor keeps a pointer to the new object for subsequent transaction delegation.
3. RW_Mdarts() constructor returns.

The contract specified when an MDARTS object is originally created is more significant than contracts checked by existing objecs. The constraints in a creation request determine the database service class that will be chosen for that object, whereas subsequent contracts are simply checked against the existing object to ensure that it can meet the requirements of that contract. A typical application will maintain a set of shared object creation declarations for each MDARTS SDM to be loaded during initialization. These declarations could be read from a file and processed by the SDM before it begins accepting RPC requests. Naturally, it is important to ensure that the shared objects are created before requests for instances of existing objects are generated.

Several of the object creation steps for shared objects manipulate class type information. For example, the SDM sometimes needs to check for type compatibility in construction requests. Furthermore, the SDM needs to pass the type of the database service class back to the client task. To perform runtime type comparisons and class lookups, MDARTS includes an implementation of runtime type information for its classes similar to that described by Stroustrup [85]. Each MDARTS class registers itself at initialization time with its runtime type information as a key. Two class registries are maintained by MDARTS, one for interface classes and one for database service classes. Database service classes also separately register their exemplars (see Chapter 3) with the interface classes from which they inherit.

2.9 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. This "embedded SQL" is converted into C by the preprocessor and subsequently compiled into an executable program. Mixing query languages and compiled languages creates the well-known "impedance mismatch" problem of database applications. In essence, the problem is that query languages were designed to provide very high-level operations on the database, and functions written in compiled languages often need lower-level access to the data elements. 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 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. Our goal is to make the database accesses as natural and as convenient as possible for a C++ application programmer.

2.9.1 Design

Figure 2.16 illustrates the MDARTS C++ application programming interface (API). Two MDARTS classes corresponding to the same database object are shown in Figure 2.16: MdartsArray<T> and ReadOnlyMdartsArray<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 manage arrays of Point objects in Figure 2.16 can also manage arrays of other types of data objects. Thus, with template instantiation, new data structures designed by application programmers can be added to the MDARTS database library very easily.

The "exclusive_update" constraint specified by the sensor task in Figure 2.16 causes MDARTS to reject subsequent attempts to construct objects that could modify the data. This constraint allows the MdartsArray<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 [90], MDARTS can perform concurrent read and write transactions without locking the data. This technique relies on the restriction that only one write transaction will be active at a given time. The "exclusive_update" constraint guarantees that this will be the case. It is important to note that constraints such as "exclusive_update" are checked only during initialization of the data objects. Sub-

```
/*****************************************************************
*  Declaration of MDARTS object in sensor task that will be updating it:
*/
MdartsArray<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:
*/
ReadOnlyMdartsArray<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 2.16: MDARTS C++ application programming interface.

sequent database access using the objects is not burdened with the overhead of checking access permissions. In Figure 2.16, the control task declaration specifies its object as a ReadOnlyMdartsArray<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 implement the ReadOnly semantic restriction as a separate class so that such errors can be detected by the compiler. This is an exception to our rule of keeping semantic constraints out of the class names. If "read-only" were only specified in the contract string, illegal update attempts would not be caught until runtime.

Interestingly, the API of Figure 2.16 does not seem to correspond to the get/setValue() interface shown in Figures 2.3 and 2.6. This is because MDARTS uses the operator overloading features of C++ to simplify the programming interface to the database. MDARTS also uses type conversion operators and temporary classes to effectively overload methods based on their return types (this issue was discussed in Section 2.4.2). The MDARTS application programming interface is remarkably simple, when one considers that it is achieved in standard C++. There are no preprocessing stages required to translate this elegant syntax into some compilable intermediate form. Therefore, MDARTS avoids the debugging difficulties introduced by preprocessing stages.

2.9.2 Implementation

The MDARTS API is deceptively simple. Suppose a task has created an MDARTS object called db_object. Now consider the following statement:

```
int i = db_object("size") + 5;
```

This statement needs to be translated into the following equivalent sequence of statements:

```
int temp = db_object.getIValue(Base::unspecified,"size",0);
int i = temp + 5;
```

One approach for implementing this would be to overload db_object's function call operator() to call getIValue() and return that result. However, then the function call operator would not be available for returning db_object members with non-integer types. Furthermore, it would be impossible to use the function call operator for update transactions as in:

```
db_object("size") = 6;
```

This statement must be translated into:

```
db_object.setValue(Base::unspecified,"size",0,6);
```

To support this convenient syntax for both read and update transactions, MDARTS overloads the function call operator() to return a temporary object that actually performs the calls to get/setValue(). There are actually two classes of temporary objects, one for read-only MDARTS objects and one for read/write MDARTS objects. The read-only class, simplified slightly, is shown below. Note that the original expression: db_object("size") is first converted to an MdartsElement object by ReadOnlyMdarts::operator(). Next, the temporary MdartsElement object is converted to an integer with MdartsElement::operator int(). This operator calls handle.getIValue(itag,tag,index). Since the MdartsElement constructor made the assignments itag = Base::unspecified and index = 0, this getIValue() call is converted to: getIValue(Base::unspecified,"size",0) for that ReadOnlyMdarts object. The read/write MDARTS objects use the same method except that their RWMdartsElement objects have overloaded opertor =() operators to call the setValue() methods of their MDARTS objects.

```
1  – class MdartsElement {
2  – public:
3  –     MdartsElement(ReadOnlyMdarts& b, const char * s) : handle(b),
4  –         itag(Base::unspecified), tag(s), index(0) { }
5  –     inline operator int()    { return handle.getIValue(itag,tag,index); }
6  –     inline operator double() { return handle.getDValue(itag,tag,index); }
7  – protected:
8  –     ReadOnlyMdarts&     handle;
9  –     const char *        tag;
10 –     int                 itag;
10 –     int                 index;
11 – private:
12 –     void operator=(const int& val) { }
13 – };
```

Figure 2.17: MdartsElement class for MDARTS API syntax.

Objects of type MdartsElement are created by operator() of the ReadOnlyMdarts class:

```
ReadOnlyMdarts::MdartsElement operator() (const char* str) {
   return MdartsElement(*this,str);
   }
```

Therefore, when the statement: int i = db_object("size"); is encountered, a temporary object of type MdartsElement is constructed, a reference to db_object is stored into handle, and the parameter "size" is stored into the tag member of the temporary MdartsElement object (by the constructor, shown on lines 3 and 4 of Figure 2.17). Next, the type conversion operator MdartsElement::int() of line 5 is called, since the context of the integer assignment causes the compiler to try to convert the MdartsElement object into an integer. It is this type conversion operator that actually calls getIValue() to perform the database transaction. Notice that the "=" operator is declared as a private class member. This allows the compiler to generate an error if an attempt is made to use a read-only object in an update transaction such as:

db_object("size") = 6;

For read/write MDARTS objects, operator() returns a RWMdartsElement object that has its '=' operator defined to call the MDARTS object's setValue() function. This permits read/write MDARTS objects to appear on either side of an assignment statement.

The introduction of temporary MdartsElement objects yields very convenient syntax for database queries and updates, but in a real-time database it is important to consider their effect on performance. To investigate this question, we ran some simple timing experiments using our Sun Sparc version of MDARTS. First, we measured the transaction times for a simple class like MdartsInt that returns a single integer value from shared memory. We compiled the code without optimization and timed both int val = db_object.getValue(Base::unspecified,""); and int val = db_object();. On average, the former call took 11 microseconds and the latter took 23 microseconds. Therefore, the creation of the temporary object added 73% overhead to a very simple MDARTS transaction. For more complex transactions, the 8 microsecond overhead would be a smaller fraction of the total transaction time, but one still might want to avoid using this syntax for time critical transactions. However, we next recompiled the code with optimization turned on (using the -O flag to the C++ compiler). When we ran the experiment again, the getValue() transaction took on average 5 microseconds and the operator() transaction took only 6 microseconds. Clearly, the optimizer was able to eliminate most of the overhead of creating the intermediate MdartsElement object. From this experiment, we conclude that with a good optimizing compiler, it is possible to have the syntactic convenience of operator() without paying a significant performance penalty at runtime.

2.10 Concurrency Control in MDARTS

Concurrency control is a critical issue in any transaction processing system. The consistency preservation and isolation properties of transactions capture the essence of concurrency control: the correctness of a transaction's execution should be independent of any concurrent execution of conflicting transactions. A conventional database system usually provides concurrency control by implementing the two-phase locking protocol. Two-phase locking is popular since it is easy to implement and guarantees serializability, which is the most widely accepted definition of correct concurrency control. As we discussed in Chapter 1, two-phase locking is not very well suited to real-time concurrency control. Therefore, many researchers have investigated alternative concurrency control strategies for RTDBSs.

2.10.1 Design

MDARTS can support multiple concurrency control protocols by encapsulating concurrency control in the implementation of the object methods that perform transactions. This approach, which is similar to that of DiPippo and Wolfe [21], permits each database class to use whatever protocol best fits the semantics of the data it manages. Through exemplar-based object construction, MDARTS can also match application-specified semantics with concurrency control strategies of different database classes.

We were primarily motivated to use object-based concurrency control to avoid the overhead of client-server communication. Without a database server to manage concurrency, the database objects must supply their own concurrency control. In other words, the database objects should be atomic data types [92, 73]. An atomic data type is essentially a class whose methods guarantee serial behavior in the presence of concurrent requests. Since the concurrency control can be individually tailored according to the semantics of the class member functions, it is possible to achieve higher levels of concurrency than with traditional read-write locking [21, 72, 93]. It is also possible to implement atomicity in a base class and inherit this property in derived subclasses [20]. MDARTS shared-memory objects differ from atomic data types and semantic concurrency control techniques described in the literature in that MDARTS objects are fragmented across multiple separate processes. The shared data structures, including lock information needed to synchronize access, are the only parts of the objects kept in shared memory. Despite this implementation difference, it is usually very easy to adapt semantic and object-based concurrency control techniques described in the literature to MDARTS.

To implement concurrency control in MDARTS we propose the following principles. Our current implementation reflects all of these ideas except data versioning.

Avoid unnecessary locking. When possible, use data versioning [39, 79] or multiple data copies [90] to permit concurrent read and write operations without locking.

Match locking granularity with data semantics. This ensures that locking does not unnecessarily restrict concurrency. Sha *et al.* [66], Badrinath and Ramamritham [6], and Son [79] all propose locking only the data affected by a transaction. However, identifying affected data and locking only those data are non-trivial problems in conventional database systems, where the data affected by a transaction are determined during query processing at runtime. MDARTS simplifies this problem since semantically-related data are grouped into database objects. Each object implements the methods that perform transactions on its data, so it is easy to match locking granularity with transaction semantics.

Control locking duration. Transactions should have well-defined critical sections, and lock acquisition and release should be performed within each operation that accesses a guarded resource.

Prevent unconstrained priority inversion. Priority inversion can be limited by disabling task preemption during short transaction critical sections. For transactions with long critical sections, it may be possible to avoid priority inversion by avoiding locking with data versioning techniques. Alternatively, a priority inheritance protocol such as the Priority Ceiling Protocol can be used to limit priority inversions.

Match locking methods to critical section size. Short critical sections are executed so quickly that the overhead of complex locking protocols can degrade performance. For example, a simple FIFO queue lock can perform better than a lock that uses a priority queue if critical section times are relatively short compared to the extra queuing time for the priority queue.

Although the MDARTS architecture does not specify any particular concurrency control protocol, our prototype implementation includes several types of lock objects. Lock objects are kept in shared memory along with the data they are guarding (see Figure 2.4). Database object methods that require mutual exclusion can acquire locks before performing their critical sections. The semantics of the transaction determine whether and for how long locks are needed. For example, a fixed-size array might support a transaction that retrieves the array size. Since the size of the array will not change, there is no need to acquire a lock to read it. However, suppose each array element contains a compound structure that requires multiple bus operations to store or retrieve. In this case, mutual exclusion must be provided for transactions that read or update array elements. Transactions that require more complex processing, such as summing the array elements, can require the entire array to be locked for the duration of the transaction. If a lock is associated with each array element, greater concurrency is possible, but the locks will consume memory proportional to the array size. Acquiring multiple locks for complex operations on an object (e.g., summing an array) can also add substantial runtime overhead compared to acquiring a single lock. MDARTS does not dictate an implementation policy on such issues. Instead, it is deemed the responsibility of the database class designer to implement concurrency control strategies that are appropriate to the data and application semantics. The performance and resource costs of the class implementations will be reflected in the timing and resource constraints that the class can support.

In MDARTS, we assume that the distribution of tasks sharing an object (and hence its lock) may not be known until runtime. The alternative is to require full knowledge of all resource sharing in the system prior to runtime. In complex systems, it is unrealistic to expect such complete knowledge. Therefore, MDARTS registers information with its lock objects during initialization to make transaction-time guarantees.

2.10.2 Implementation

Locking delays are the primary source of transaction-time uncertainties for memory-based MDARTS objects. Therefore, we have designed locking strategies in our prototype implementation that allow MDARTS objects to bound locking delays for their methods and thus determine overall transaction-time guarantees. We will discuss how locking delays are bounded by one of our MDARTS locks, a spinlock queue.

Recall from Section 2.7.2 that transaction-time guarantees depend on D, the worst-case blocking delay for a transaction to acquire a lock. For a given lock, D is bounded by $QTime + (LCScount \times LCStime)$, where $QTime$ is the overhead required to enqueue and dequeue a transaction, $LCScount$ is the worst-case number of critical sections that a transaction may have to wait before it acquires the lock and $LCStime$ is the worst-case execution time for critical sections guarded by the lock. $LCStime$ depends on the implementation of transactions that use the lock. As each object initializes, it registers its worst-case critical section time with its lock object. This critical section time is measured during object benchmarking, as described in Section 2.7.2. $LCScount$ depends on the locking and queueing protocol implemented by the lock and upon the number and distribution of objects sharing it.

We have implemented a spinlock queue lock that uses a simple FIFO queueing strategy. The literature contains several examples of implementing such locks on shared-memory multiprocessors, e.g., [3, 19]. A transaction requests a lock by invoking the lock's `getLock()` method. Just before enqueuing a transaction, `getLock()` disables task preemption. This means that while executing the critical section or while waiting for the lock, the transaction effectively acquires the highest execution priority in the system. It is necessary to prevent preemption so that tasks will not be delayed if tasks ahead of them in the queue are preempted. On a multiprocessor, it is especially important to ensure that once transactions enter a spinlock queue they are not preemptible. This is because transactions executing on different CPUs could spin for an unbounded time if a remote task ahead of them in the queue is preempted.

If preemption is disabled and a FIFO queue is used, it is easy to determine $LCScount$, the maximum number of critical sections a transaction would have to wait before acquiring a lock. By disabling preemption, we not only bound the time required for each transaction to complete its critical section and release the lock, but we also ensure that no more than one transaction from each CPU can enter the queue. Therefore, for any number of tasks distributed across m CPUs and sharing a lock, $LCScount = m - 1$. The benefits of disabling preemption do have a cost, however. The maximum time a transaction might disable preemption must be considered when analyzing task schedulability. If this time is too long, the schedulability of tasks with tight deadlines could be jeopardized. Therefore, the maximum synchronization delay D for any lower-priority tasks with locks that disable preemption must be included as a component of priority inversion for the higher-priority tasks. If these delays are bounded tightly enough, it will still be possible to guarantee task schedulability for the application.

Each task that declares a shared object containing a spinlock automatically registers with that lock during the initialization of its object instance. To register with a lock, the MDARTS object indicates the execution time of its longest critical section and the shortest slack time for its timing constraints. The longest critical section is usually fixed for a given object, but it could vary due to differences in CPU speeds. The shortest slack time is the smallest difference between the application-specified timing constraints for a transaction that uses the lock and the execution time of that transaction when no locking delay is experienced. If a transaction has multiple critical sections, the slack time registered with the lock is divided by the number of critical sections. As tasks register new instances of shared objects, the lock keeps track of the minimum slack time registered thus far. The lock also counts how many different CPUs have registered with it. If a task on a new CPU attempts to register, but that task would cause a violation of a previously-guaranteed transaction time, the registration attempt is rejected by the lock. A transaction-time guarantee is violated if the D that would result after the new registration exceeds the minimum slack time.

Clearly, spinlock queues are appropriate only for very short critical sections. In our MDARTS implementation, we have focused on applications whose data needs match this description, namely high-speed control systems. For synchronization techniques applicable to longer critical sections, refer to Chapter 4.

2.11 Current Status

Currently, we have completed implementation of the MDARTS framework with the Shared Data Manager, local and remote updates, and exemplar-based object construction using semantic and timing constraints. We have also implemented shared lock objects for concurrency control. MDARTS currently runs on VxWorks-based shared-memory multiprocessors and on Sun workstations using System V shared memory and socket communication.

CHAPTER 3

CONTRACTS

3.1 Motivation

Several times in Chapter 2 we mentioned that MDARTS dynamically constructs database objects according to application semantics. We now explain this capability in the context of object-oriented library design and present the implementation approach we have taken in MDARTS to achieve it.

Object-oriented programming features such as polymorphism, encapsulation, and inheritance, encourage the development of sets of classes that are similar in functionality but are customized in various ways. Software libraries populated with such classes, such as the MDARTS library, can be made both flexible and efficient, thus encouraging reuse. Each customized class in the library might support different space and time efficiencies for member functions and various combinations of features such as range checking, persistence, and concurrency control. For example, the library of generic container classes supplied with the gnu C++ compiler includes eleven customized Set classes, each using different underlying data structures and algorithms. Application writers must understand the subtle differences between customized classes in the library to select the best class and use it correctly. As libraries become larger and more complex, this problem becomes increasingly difficult.

In real-time applications, the subtle differences between similar classes in a library can be very significant since memory is likely to be more limited than in a conventional application, and timing differences between different implementations will affect the application's schedulability. Therefore, the space-time tradeoffs of alternative class implementations are more likely to become critical issues in real-time systems. Class browsing tools are often proposed to assist application writers in selecting classes [33], but these tools still expose the full complexity of the class hierarchy. Applications that use specific subclasses in a customized class hierarchy can become dependent on the internal class structure of the library. Such dependencies make future reorganizations of the library classes difficult to accomplish without propagating changes to existing applications. Therefore, applications should be kept as independent as possible from the structure of a library's internal class hierarchy. Class browsing tools do not address this need.

Furthermore, since many of the differences between customized classes may reflect semantic differences that are not expressible in the syntax of the language, there is a danger of mismatch between the semantics supported by a customized class and its actual use in an application. Meyer discusses this problem and shows how software contracts in Eiffel can detect some types of semantic mismatch at runtime [52]. Although Eiffel software contracts help check consistency, they do not help select the server object in the first place. Furthermore, the runtime contract checking in Eiffel adds overhead and slows method execution.

In this chapter, we describe a novel approach to software contracts that uses explicit application-side contracts and exemplar-based programming techniques to 1) automatically determine which customized server object to create and 2) configure the server according to application needs. Our object construction mechanism also improves encapsulation by hiding part of the library's internal class hierarchy from applications. MDARTS uses this exemplar-based technique to customize its database services at runtime, according to the real-time and semantic characteristics of applications. To our knowledge, no other real-time database system has comparable dynamic adaptability to application requirements.

The remainder of this chapter is organized as follows. Section 3.2 reviews prior work on software contracts and introduces our approach. Section 3.3 discusses customization and ways application-side contracts can simplify the selection of customized server classes. Section 3.4 presents the MDARTS design, which uses application-side contracts with exemplar-based programming to implement the server class selection discussed in Section 3.3. Section 3.5 discusses the MDARTS implementation of contracts. Section 3.6 concludes and discusses future work.

3.2 Software Contracts

By analogy to contracts employed in civil law, software contracts have been proposed to formalize relationships between software entities. The software entities could be two interacting processes, an application and a software library, a server object and a client object or application, or a base class and a derived class. Although several researchers have used the contract metaphor, there is little consensus on what a software contract should specify or how it should be expressed.

Wirfs-Brock *et al.* define contracts to be the set of methods exported by a server object [94, 95]. In this case, the contract is pure metaphor: a useful perspective on existing structure. The Eiffel language provides support for a more tangible form of software contracts [52]. Contracts in Eiffel are constraints on the pre- and post- conditions of functions. These constraints are assertions that check at runtime if applications are using functions correctly. Eiffel also supports the related concept of class invariants, which can be considered contracts specifying consistency between base classes and derived classes in an inheritance hierarchy. Helm *et al.* propose a higher-level use of the contract metaphor in which contracts specify roles and interactions between cooperating objects [33]. Applications instantiate contracts at runtime by selecting classes for the various roles.

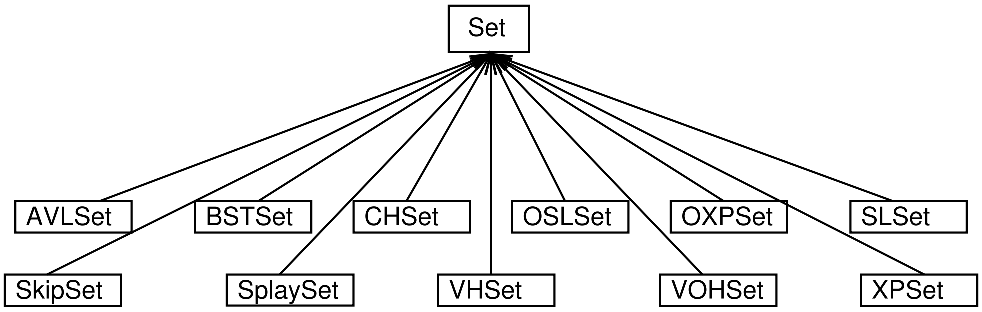
Like the software contracts described by Wirfs-Brock *et al.* [94] and Meyer [52], our contracts apply constraints to individual objects rather than behavioral compositions as in Helm *et al.* [33]. However, since the application does not necessarily specify the exact class to which the contract applies, our contracts are more accurately viewed as being between the application and the software library. Furthermore, we distinguish between explicit software contracts and implied software contracts. The implied part of a contract corresponds roughly to the contracts in [52, 94]; the explicit part is the subject of this chapter. We base our approach to software contracts on the following analogy with contract law. When a legal contract is established between a service provider and a client, there is both an express and an implied contract. The express contract is comprised of the specific clauses in the contract document. The implied contract consists of the reasonable and customary duties of that kind of service provider. For example, a contract with a plumber might specify the brand of faucet to install in a kitchen. If the plumber installs the right type of faucet but the plumbing leaks, the plumber is liable for damages even if the contract does not specifically mention leaks. This is because a plumber's professional duties routinely include leak-free installation of plumbing. A leaky installation is a violation of the implied contract.

We consider the methods exported by a class and any class invariants to constitute an implied contract between the server class and the application. By selecting a server class, an application establishes the implied contract with the server that covers most aspects of its subsequent use. However, just as in legal contracts, the application may need to specify explicit terms that must be fulfilled in addition to the implied terms. It may be that only certain specialized server classes can satisfy the explicit terms. In this case, the contract can be used to select an acceptable class from the general population of server classes. In other cases, the explicit terms might relax certain constraints and thereby permit server objects to optimize various aspects of their services. For example, a server object that supports concurrent access could use simplified locking protocols if it knew the application would not perform concurrent update operations. This semantic constraint could be supplied by the application in the contract.

Wirfs-Brock *et al.* [94] and Meyer [52] focus on contracts defined by server objects. Since the server dictates all the terms, there is no way for applications to add clauses or establish their own contracts. Applications must know at compile time which server classes support the desired functionality, a requirement that becomes especially burdensome when the library contains many similar server classes. By contrast, in our system, applications can create explicit contracts to specify requirements and communicate application-dependent semantic information to servers. Contract clauses can contain either specific constraints or preferences. Specific constraints, such as "persistent," are either met or not met by a server. Preferences, such as "minimize_execution_time," indicate the characteristics most important to the application. Preferences can be used to guide the selection of servers and to help servers configure their services to application needs. Just as in legal contracts, explicit application-side software contracts are complementary to implicit server-side contracts. In the remainder of this chapter, we restrict our discussion to explicit software contracts, which we simply call "contracts." The contracts are physically represented by character strings passed by applications to the server library. Contracts are evaluated at runtime during server object initialization. Our approach resembles service specification and acquisition in distributed computing systems [14, 63], except our server objects are much lighter-weight and are constructed from local libraries rather than remote server processes. If no server objects in the library can meet the requirements of the contract, then the library can set an error flag or throw an exception. In the next section, we discuss how application-side contracts can help determine which customized server objects will be used.

3.3 Contracts and Customization

Through customization, object-oriented programming can partially overcome the classic tradeoff between flexibility and efficiency in software libraries. Instead of supporting a single, general-purpose implementation of a function or abstract data type, an object-oriented library can provide a variety of specialized classes that collectively cover the same domain but are individually more efficient than a general-purpose implementation. Specialized classes can be more efficient since the best implementation of a given function may depend




```
VHSet<int>  application_set;  // application must explicitly choose the class
```

Figure 3.1: Generic Set classes.

on the patterns of use within the application. For example, a set class that keeps members in a hash table would perform well if the application mainly tests for set membership. However, a linked-list internal representation might be better if memory is scarce, the set contains few members, or the application adds members frequently and rarely tests for membership. Instead of choosing a single compromise implementation, an object-oriented library can contain multiple compatible classes, each optimized for certain operations. An application can then choose a customized class according to its needs. With polymorphism, the programming interface to such classes can be made convenient and consistent.

However, with the flexibility of choosing from a group of similar classes comes the burden of understanding the characteristics of each and making a good choice. There is also a danger of mismatch between any restrictions imposed by a customized class and the way it is used in an application. For example, a server object used in a multithreaded environment might use a concurrency control protocol that supports a single writer and multiple readers. If the single writer restriction is violated by the application, as could happen accidentally since the restriction may not be expressible through language syntax, the object is likely to become corrupted.

The usual approach to choosing a customized class is to instantiate the class by name. The customized characteristics are encoded in the class name. For example, the generic container class library distributed with the gnu C++ compiler contains a base class and eleven derived classes that implement sets: Set, AVLSet, OSLSet, VHSet, BSTSet, OXPSet, SkipSet, VOHSet, CHSet, SLSet, SplaySet, and XPSet. Each name encodes the data structure and algorithms the class uses to implement sets. Applications choose the set implementation by using the corresponding class name. While this technique for specifying customization is easily understood, in practice it is unwieldy. As additional semantic attributes such as persistence or concurrency are added, the class names either become very long or very cryptic. It also becomes difficult to remember the correct order for semantic attributes in class names. Even worse, each combination of semantic attributes implies a unique class. Thus, the addition of new semantic attributes results in an exponential explosion in the number of classes. If persistence and two types of concurrency control (e.g., single-writer and multiple-writer) were added to the gnu Set classes, the eleven subclasses would become sixty-six (persistent and non-persistent versions of the original classes plus persistent and non-persistent versions for each type of concurrency control). Figure 3.1 illustrates the class hierarchy for the Set classes and how an application creates an instance of a Set class.

Because selecting an appropriate customized server class is so dependent upon application-side semantics, software contracts specified by applications can aid in the selection process. Instead of using class names directly, applications can use contracts to provide a mapping between application requirements and server classes. With this more flexible means of communicating semantic requirements, it is possible to avoid unnecessary proliferation of classes by supporting several combinations of semantic attributes in a single class. For instance, a single concurrent implementation of a class could support several locking protocols for single-writer or multiple-writer semantics.

In MDARTS, contracts are composed of constraint clauses encoded in character strings. The syntax of the contract language is implementation-specific. A typical contract string might be: "range_checked; lookup_time<=O(log n);". Using strings for contracts is simple, portable, convenient, and offers more flexibility than plausible alternatives such as defining language extensions for processing contracts at compile time. An application may need to dynamically determine contract constraints at runtime, so compile-time contract processing is not always possible. Furthermore, by leaving contract interpretation to class member functions rather than embedding it in the compiler, we preserve the ability to define new types of constraints with whatever syntax is most convenient in the context of a particular class. It is even possible to define new constraints in customized subclasses without modifying the base class. This provides great flexibility and helps keep contract constraints orthogonal to the class hierarchy. Another reason we chose to use character strings for contracts is that MDARTS performs remote object creation by passing contracts across the network using remote procedure calls. Character strings are easy to use for RPC parameters.

The contract string is supplied as a parameter to a server class constructor function. The server class constructor examines the contract string during object initialization and either configures a new object to meet the terms of the contract or rejects the construction request. An application can specify the exact class desired, or it can specify one of the base classes in a hierarchy of customized classes. In the latter case, exemplar-based programming is used to automatically select the customized class according to the contract. Section 3.4 describes our use of exemplar objects in more detail.

For example, an MDARTS application might declare a persistent array object as follows: `MdartsArray<int> parts_list("parts_list","persistent; range_checked; sparse; size=1000")`. In this case, the base class MdartsArray is specified along with the name of the object and its contract string. Some constraints, such as "persistent", might be supported only by specialized subclasses that access a disk-based database system. Some, such as "size", are supported by all MdartsArray subclasses and help initialize the state of the object. The size constraint would be implemented in the base class and be inherited by subclasses. Others, such as "sparse", might be used to determine the subclass or might just set a flag in the object during initialization. The mapping of contract constraint clauses onto the subclasses of the MdartsArray class is of no concern to the application. The library either will construct a customized, correctly-configured server object or will signal an error (return NULL or raise an exception).

There are many advantages to customization through software contracts. One advantage is that application-specific semantic attributes can be explicitly stated in object declarations. This is especially helpful when the attributes represent hints to server objects that enable various optimizations. By declaring these hints, the application is expressing a willingness to abide by whatever restrictions are implicit in these hints. For example, single-writer concurrency control protocols can reduce locking delays compared to more general concurrency control methods [91]. However, a server object cannot control application behavior to ensure that the single-writer restriction is followed. With contracts, a single-writer server class would not be chosen unless the application explicitly indicated in the contract that it would avoid concurrent updates (by specifying a concurrency constraint such as "exclusive_update"). For software reliability and maintenance reasons, it is important that such restrictions be declared in the code itself rather than existing only in documentation or in the mind of the application developer.

Another important advantage of customization through contracts is that it allows applications to maintain a simplified view of the library's class structure. We call this "class hiding" because the class hierarchy beneath each abstract server base class is hidden from applications. The MdartsArray example discussed above illustrates this idea. There could be dozens of different classes in the library that support the abstract interface defined in the MdartsArray base class. Because the semantic attributes that determine which subclass to use are passed in the contract, the application can safely ignore the underlying complexity and still be assured that a server customized to its needs will be created. If no server can be created, the library can signal an error condition (throw an exception). The server subclasses in each class hierarchy can be viewed as a population of contractors with the same trade. Each has different abilities and specialties, but all follow the same basic protocols. The application need not be familiar with each contractor's idiosyncratic combination of specialties. Instead, the application creates a contract and submits it to the population of contractors. The techniques we use to accomplish the contractor selection process are described in the next section.

Naturally, there are also disadvantages to using application-side contracts, and the method should not be applied indiscriminately. Contracts add complexity to the library implementation, and they impose runtime overhead as they are evaluated during server initialization. Contracts may not be worthwhile for libraries with few customized class hierarchies. The runtime overhead during initialization is amortized over subsequent server use, so the efficiency gained through customization in server functions must exceed the initialization overhead for the method to be worthwhile. If an application wants to avoid the overhead of processing contracts during initialization, it always has the option of explicitly specifying a server class. Doing so will lose the benefits of class hiding and semantic checks, but that decision can be left up to an application. In general, contracts are best for server objects that will be used frequently by applications and that can achieve significant performance improvement through customization. However, even for cases in which efficiency gains through customization cannot justify the use of contracts, contracts may still prove useful for specifying semantic constraints to reduce errors in server usage.

Another limitation of our contract implementation is that errors in contract strings are not detected until runtime. This problem is unavoidable if dynamic creation of contract strings is allowed. Finally, since it is unknown until runtime which server classes will be used, application executables might become very large as they incorporate all of the customized classes. We address this issue in the next section. Although contracts are not necessarily useful in all circumstances, a library need not support contracts for every class hierarchy. Support for contracts can be embedded in only those hierarchies where they are needed.

3.4 Exemplars and Customized Classes

Exemplar-based programming, in which prototype objects play a role similar to that of classes, is often cited as an alternative to more traditional object-oriented architectures. The best known example of this approach is Self, a language which uses exemplars and delegation to dispense with classes altogether [89]. While exemplars in Self form the basis of a complete programming paradigm, exemplars can be useful in a class-based object-oriented context as well. Coplien illustrates the use of exemplar-based programming in C++ [16]. In our implementation, we combine software contracts with Coplien's autonomous generic exemplar idiom (in which exemplars register themselves with a base class and object construction requests iterate over the exemplars).

Exemplars are special, one-per-class objects that are prototype representatives of an entire class. Given an exemplar object, applications can construct copies of the exemplar by invoking a special clone() method. Because exemplars are objects, they can be stored in data structures. In some object-oriented languages, classes are first-class objects, so class objects could be used with our technique instead of exemplars. We put exemplars (or class objects) in data structures to manage collections of customized server classes. These representatives comprise the populations of contractors mentioned in the previous section. Rather than choosing a specific server class, an application chooses a base class and specifies the rest of its requirements in a contract string. The contract is passed to the population of contractors (exemplars) derived from that base class. The exemplars then bid on the contract to determine which class meets the application's requirements. The winner of the bidding process is cloned, and the clone object is returned to the application.

All classes (exemplars) in each contractor population support the same functional interface, so applications can use the cloned object without knowledge of which class was actually constructed. Therefore, applications only contain dependencies on the abstract base classes in the library. This is a very significant point, because it permits class libraries using our techniques to be extended, modified, and reorganized without requiring corresponding modifications to existing applications that use the library. The concept of encapsulation is thus extended to include encapsulation of entire class hierarchies.

For example, a library might initially contain an array base class and two customized subclasses: one that supports concurrency and one that does not. Each of these subclasses would contain various configuration options to support different combinations of semantic attributes. Suppose applications using this library are developed. Now suppose that the library developer decides to split the class that supports concurrency into three separate classes, each customized to support a subset of the attributes supported by the original concurrent class. The developer may want to do this to improve efficiency. As long as applications use the array base class and specify semantic requirements in contracts, multithreaded applications whose contracts originally mapped to the single concurrent class will now automatically use one of the new classes. No source code modifications in the applications are required. The programs need only be relinked with the new library.

One of the key advantages of using exemplars is that it facilitates extensibility. New classes can add their exemplars into the appropriate exemplar collection, and no existing library or application code need be changed to accommodate the new class. Without the exemplar mechanism, existing applications can benefit from new classes only if the application code is changed to specify the new classes or an object-creation function in the library is modified to include the new classes.

Given exemplar-based object construction, there are still numerous implementation issues to consider. For instance, how are the exemplars organized?, how is the bidding process accomplished?, how can applications avoid linking in unneeded exemplars?, etc. In the remainder of this section, we consider these issues. In Section 3.5, we present an example C++ implementation of contracts and exemplar-based object construction.

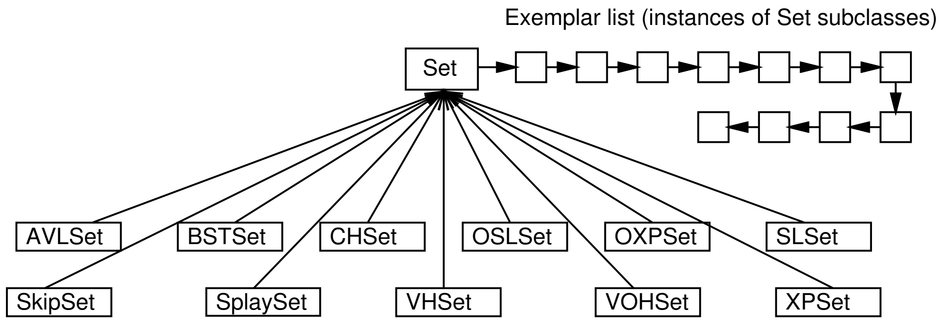
Since we are interested in groups of exemplars derived from a common base class, the data structure containing the exemplars should belong to the base class. The simplest way to do this is to create a linked list for each server base class. Exemplars of derived classes are added to the list during exemplar initialization. Bidding can then proceed by iterating over the list and submitting the contract to each exemplar in turn. Either the first exemplar to satisfy the contract is cloned or the exemplar that best satisfies the contract is cloned. In the former case, the iteration proceeds until one of the exemplars clones itself. In the latter case, each exemplar returns a "bid" value in response to the contract. The function performing the exemplar iteration keeps track of the most attractive bid and clones that exemplar once all of the bids are examined.

Clearly, if large numbers of exemplars are associated with each base class, iteration over all of them will be slow. If the first exemplar to satisfy the contract is cloned, performance will be somewhat better. However, in this case the order of exemplars in the list may influence which server class is constructed. Since applications might prioritize different server characteristics, there may not be a single list ordering that is best for all applications. Nevertheless, the "first bid wins" approach does ensure that the server returned will satisfy the requirements specified in the contract.

To improve the performance of the bidding process, one could use more sophisticated techniques than iteration over a linked list. The selection of a server can be viewed as a search process over exemplars using the contract as the key. If the exemplars are organized during library initialization into a classification network or a signature-based hash table, the search could be guided at runtime by the contract. If there are many exemplars, this could dramatically reduce the number of exemplars asked to bid on a contract. However, the value of complex algorithms must be weighed against their cost. More complex data structures require more memory and more complex search algorithms. Since the library maintains multiple exemplar lists, each attached to a different base class, most of the exemplars in the library are eliminated from consideration when the application specifies the base class. Since the exemplar bidding process is performed only during server object construction, it is unlikely to occur inside tight application loops where efficiency is crucial. The best technique for exemplar bidding ultimately depends upon the particular class hierarchy and expected patterns of use by applications. This is an interesting research problem on its own.

A naive implementation of contracts and exemplars would include all exemplars (and their associated code) in applications using the library. If most of these exemplars are never used (cloned) by an application, which is likely, this means lots of unused code will be linked into the application. Furthermore, the presence of unused exemplars will slow the bidding process. Ideally, one would like to have each exemplar list contain only those exemplars that will be used by the application. Unfortunately, this information is not known until runtime.

If the contracts do not depend on runtime information, so the same exemplars are always cloned, this problem can be solved with the following technique. For each server base class there is a header file that causes all customized exemplars to be linked into an application. Applications during development and testing use this header file so all exemplars are included. If a certain mode is enabled during testing, the base class keeps



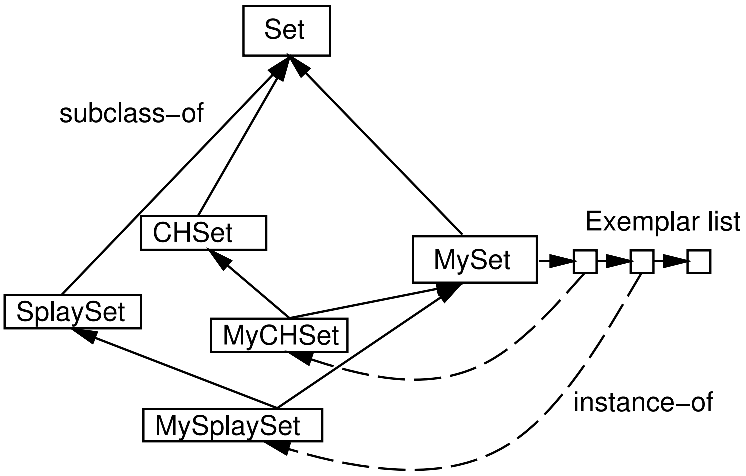

```
Set<int>  application_set("space=O(n);time=test_membership<=O(logn);prefer=time");

// note that with contracts, applications declare their needs directly
// in the contract rather than implicitly through choosing a specific
// class from the library.
```

Figure 3.2: Generic Set classes with exemplars.

track of which exemplars are actually cloned in each application run. As the application terminates, the base class exemplar (in its destructor) writes a new header file that includes only the classes of cloned exemplars. Applications ready for production can use the new header files and thereby avoid linking in unused exemplar code. With this technique, the exemplar-based approach to customization does not necessarily lead to bloated code size. However, as memory prices fall and operating systems add support for shared libraries, code size becomes less important. Therefore, in the long run it will probably be unnecessary to eliminate unused exemplars.

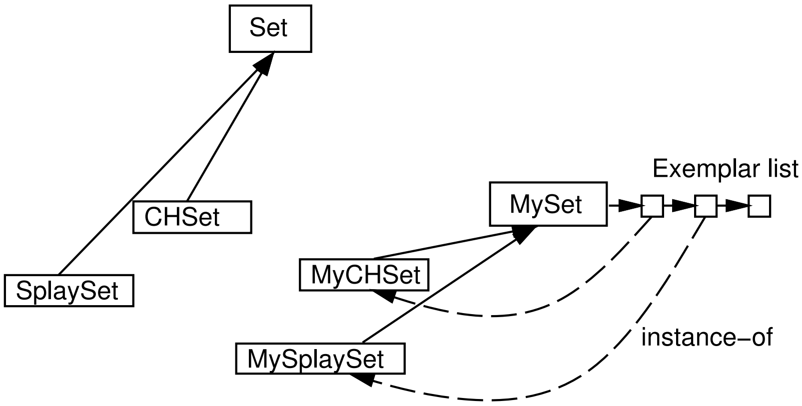
Figure 3.2 shows how exemplars and contracts could be used to simplify the application interface to the generic Set classes introduced previously. The object declaration below the diagram in Figure 3.2 shows that applications do not need to specify a particular server subclass if exemplars are used to select the servers. Instead, the application can use a single template class, Set<T>. This example raises an interesting question: how difficult is it to add support for contracts to an existing class library? If one has access to the library source code, it is possible to add constraint-checking methods and exemplar objects to the existing classes.

However, what if it is not feasible or not desirable to modify the library source code? Our basic contract and exemplar techniques would not work in this case, since the exemplars are instances of the classes, and they must support the constraint-checking methods to bid on contracts. There are two alternatives to modifying existing library code. One possibility is to create a new class hierarchy derived from the original library using multiple inheritance to add the necessary methods. This approach is illustrated in Figure 3.3. The class MySet in Figure 3.3 contains the exemplar list and the methods for checking contracts and selecting exemplars. The new subclasses such as MyCHSet inherit and override the constraint-checking functions from MySet and also inherit the Set functions from the original library classes. Applications use the cloned exemplar objects through the MySet interface. A second approach is to create a shadow class hierarchy that simply encapsulates knowledge of the existing classes but does not inherit from them. The exemplars of the shadow hierarchy process the contracts and determine if the original class they represent would satisfy them. When one of these shadow exemplars is chosen, it creates a new instance of its original class instead of cloning itself. Figure 3.4 illustrates this approach.




```
MySet<int>  application_set("space=O(n);time=test_membership<=O(logn)");
```

Figure 3.3: Adding contract support via inheritance.




```
Set<int>  * application_set =
          MySet<int>::make("space=O(n);time=test_membership<=O(logn)");

// note that the application interface with a shadow hierarchy requires a different
// syntax:  an explicit call to MySet<int>::make() is required, and a pointer to
// an original Set object is returned.
```

Figure 3.4: Adding contract support via a shadow hierarchy.

```
class Base {
protected:
    // ConstructServer() parses the constraints and tries to clone an object
    // that meets them.  Returns a server object if successful, NULL otherwise.
    //
    Base * ConstructServer(const char * contract) {
        Base * server = 0;  // server object, if cloned successfully
        Base * ex;          // exemplar object in list
        Plist<Constraint> cl;
        MakeConstraintList(contract,cl);
        // scan down the exemplar list to find one that meets all constraints
        Plist<Base> & elist = getExemplarList();       // virtual fn call
        for (Pix p = elist.first(); p; elist.next(p)) {
            ex = elist(p);
            server = ex->checkAllConstraints(cl) ? ex->clone() : 0;
            }
        return server;
        }
    // Pure virtual functions below.  These methods must be implemented
    // by MDARTS server classes.
    virtual void        registerExemplar(Base * ptr) = 0;
    virtual int         checkConstraint(const Constraint& c) = 0;
    virtual void        stageConstraintCheck() = 0;
    virtual Base *      clone() = 0;
    virtual Plist<Base> & getExemplarList() = 0;
};
```

Figure 3.5: Contract and exemplar methods in Base.

3.5 Example C++ Implementation

Although our approach to contracts and customization is not language-specific, our prototype MDARTS library includes a C++ implementation of exemplar-based customization. In this section, we present this implementation. We first discuss support for constraint processing in the MDARTS base class Base. We next present two example server classes: a server base class called "MdartsArray" and a customized server class called "RangeCheckedArray" that supports the semantic constraint "range_checked". For brevity, part of the MDARTS class hierarchy and some class member functions are omitted.

Code common to all exemplar-based class hierarchies is factored into the MDARTS abstract base class called Base. This class contains methods for parsing contract strings and cloning server objects from lists of exemplars. Parsing contracts into lists of constraint structures was discussed in Section 2.6.2. Since Base declares pure virtual functions needed for exemplar-based object construction, derived classes are forced to implement the required functions. Figure 3.5 shows the Base class code used to implement exemplar-based object construction. This code uses the simple "first bid wins" approach for choosing the exemplar to clone.

The constraint checking function in each server class consists of a switch statement over the enumerated constraint type. Figure 3.6 lists the code of an example server class called MdartsArray. Another example of checkConstraint() is presented in Figure 2.10.

Note that class MdartsArray exports a public function called "make()" that invokes the Base::ConstructServer() function. This is the function used (directly or indirectly) by

```
class MdartsArray: public Base {
public:
   static MdartsArray *      make(const char * contract) {
      return (MdartsArray *) ConstructServer(contract);
   }
   // public MdartsArray accessor functions here. e.g., operator() ...
protected:  // methods visible only to member functions and subclasses
   // constructor for array object
   MdartsArray(int s) {
     shared = shared_memory_malloc(sizeof(shared_data)+(s−1)*sizeof(int));
     shared→theSize = s; }
   // array representation: size and first element in array, stored together
   // in shared memory.  A more realistic example would be a template class...
   struct shared_data {
      int    theSize;       // how many elements in the array
      int    theArray;      // start of array
   };
   shared_data  * shared;
   // list of derived exemplars – subclasses add their own exemplars
   // to this list by calling registerExemplar().
   static Plist<Base>   TheExemplarList;
   Plist<Base> & getExemplarList() { return TheExemplarList; }
   // Implementation of pure virtual functions defined in class Base.
   void registerExemplar(Base * ob) { TheExemplarList.prepend(ob); }
   void stageConstraintCheck() { shared→theSize = 1; }  // initialize state
   // check a single constraint
   int   checkConstraint(const Constraint& c) {
   switch (c.constraint_type) {
      case size:
         shared→theSize = atoi(c.value); // convert argument string to integer
         return 1;  // success (should verify c.value is an unsigned int)
      default:
         return 0;  // failure (reject all constraints except "size")
      }
   }
};
// definition of static (one per class) exemplar list
Plist<Base> MdartsArray::TheExemplarList;
```

Figure 3.6: Class definition for MdartsArray.

```
class Exemplar { public: Exemplar(){} };

class RangeCheckedArray: public MdartsArray {
typedef inherited MdartsArray;
protected:
   // constructors
   RangeCheckedArray(Exemplar) { registerExemplar(this); }
   RangeCheckedArray(int s) : Array(s) { }
   static RangeCheckedArray * TheExemplar;

   Base *   clone()  { return new RangeCheckedArray(shared->theSize); }

   int checkConstraint(const Constraint& c) {
      switch (c.constraint_type) {
      case range_checked:
         return 1;
      default:        // defer other constraints to base class
         return inherited::checkConstraint(c);
      }
   }
};
// definition of static (one per class) data members
RangeCheckedArray * RangeCheckedArray::TheExemplar =
   new RangeCheckedArray(Exemplar());
```

Figure 3.7: Class for range-checked array.

applications to create customized server objects. A direct use of make() by an application would look like: MdartsArray *array_ob = MdartsArray::make("size=80,range_checked");. If make() fails to create a valid MdartsArray server object (as could happen if the contract contains constraints not supported by any of the exemplars), it returns NULL. If this object construction syntax is undesirable, it is possible to encapsulate the server object pointer and the make() call in an envelope class that forwards server functions to the internal object. This is the technique used in MDARTS to achieve the application programming interface described in Section 2.9. The make() function in each server base class passes ConstructServer() the contract string and the list of exemplars of derived classes. ConstructServer() parses the contract and converts it into a list of Constraint structs. It then submits the constraint list to each exemplar until one of them returns a clone (we use the simple "first contractor to accept the contract wins" bidding technique). The Base pointer returned by ConstructServer() is type cast to a pointer to an MdartsArray object. This is a type-safe operation since all exemplars on the list belong to classes derived from MdartsArray.

Each exemplar's stageConstraintCheck() function is invoked by the Base ConstructServer() function before the constraints are checked. StageConstraintCheck() is used to initialize the state of the exemplar to eliminate carryover from prior contracts. For example, if an exemplar derived from MdartsArray processes the constraint "size=1000", it sets its internal theSize variable to 1000. This size variable is used to determine how big the clone object's array will be. If a subsequent and completely different contract is processed that does not specify the MdartsArray size, we want theSize to default to some constant value rather than retaining the value from the previous contract. Thus, MdartsArray's stageConstraintCheck() sets theSize to 1. It may be that state variables must be initialized at multiple points in the class hierarchy. Therefore, if derived classes implement stageConstraintCheck() to initialize any state specific to that subclass, before returning they should also invoke their base class(es) stageConstraintCheck() function(s). Once the exemplar state is initialized

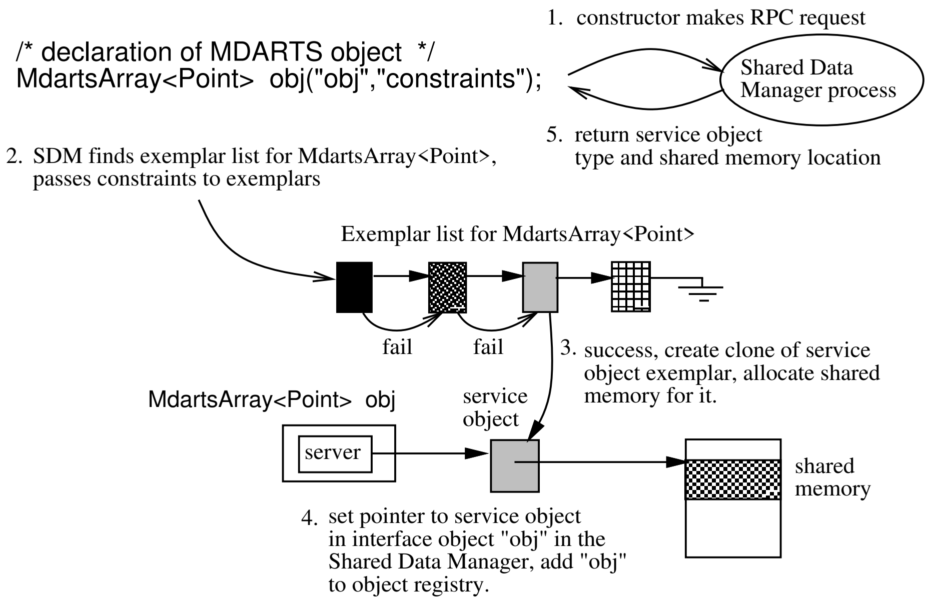


Figure 3.8: MDARTS object construction using exemplars.

via stageConstraintCheck(), each constraint in the contract is evaluated by the exemplar's checkConstraint() function. CheckConstraint() returns a boolean result to indicate whether the constraint is supported.

The Exemplar class is used as a dummy parameter to a special constructor in each derived server class that adds the exemplar to the base class exemplar list. By declaring a static pointer to the exemplar in the server class, C++ static member initialization can be used to automatically construct and register exactly one exemplar per class. This technique is borrowed from Coplien [16].

Each server class need only recognize a subset of the constraints defined by the server base class. Like stageConstraintCheck(), checkConstraint() chains up the inheritance hierarchy, deferring to its base class when unrecognized constraints are encountered (see the default clause in the switch statement of RangeCheckedArray's checkConstraint() function).

Figure 3.8 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. This SDM uses exemplar-based object construction to select and instantiate a service object that meets the application needs. After the RPC returns, the application task completes the construction of its local instance of the object.

3.6 Summary

We have described a new approach to software contracts based on application-side contract strings and exemplar-based server construction. MDARTS contracts consist of semantic constraints that are used to guide the selection of customized server objects at runtime. Our techniques permit the encapsulation of entire subtrees of classes in an object-oriented library. Instead of exposing the application developer to many similar customized classes, the library implementer can define a small set of abstract server base classes and hide the hierarchies of specialized subclasses from applications. Besides simplifying the application interface, class hiding permits radical restructuring of the library implementation without breaking existing applications. Our techniques should prove very useful for the development and management of large class libraries.

Although our exemplar and constraint checking implementation requires class library implementers to follow certain protocols, the price is not high compared with the benefits of software contracts and class hiding. Much of the complexity of our technique is localized in the Base class and in the application interface classes of each server hierarchy. The additional support required in customized server classes is nominal. Language extensions could simplify the library implementer's job somewhat by, for example, automatically providing exemplar objects and the Base functions. However, there are many practical drawbacks to creating language extensions. We prefer to use existing language features provided that they can support the desired functionality.

In the future, we plan to investigate more sophisticated techniques for organizing exemplars and conducting the bidding process. It would be useful if applications could specify both requirements and preferences (e.g., minimize runtime, minimize space utilization, etc.) to guide contractor bidding. We also intend to investigate ways to negotiate constraints with exemplars if none of the exemplars are willing to bid on the original contract.

CHAPTER 4

SEMAPHORE QUEUE PRIORITY ASSIGNMENT IN MULTIPROCESSORS

4.1 Motivation

Our prototype implementation of MDARTS focuses on memory-based transactions with very short critical sections. For transactions such as this, the overhead required to block and restart a transaction that fails to acquire a lock will generally exceed the worst-case wait time to acquire the lock. Therefore, it is more efficient to simply spin wait until the lock becomes available. The time spent spin waiting is wasted CPU time, but more than that amount of time would be lost by switching to a different task. Furthermore, if a transaction is blocked and another task is allowed to run, the new task will usually have lower priority than the first task (assuming preemptive, priority-based scheduling). This lower-priority task that is granted the CPU now has an opportunity to acquire other locks that might cause further blocking of the first transaction after it acquires the lock that blocked it. If the higher-priority task spin waits instead, it will avoid this problem. For these reasons, we have focused our implementation efforts on spinlocks.

However, the MDARTS architecture does not dictate the concurrency control policies of its objects. It is quite likely that applications will develop database objects with long critical sections for which spin waiting is highly inefficient. Any transaction that accesses relatively slow devices instead of shared memory or performs complex, expensive calculations would fall into this category. Objects with long critical sections will need to use some form of blocking on semaphores to release the CPU to service other tasks while the transaction waits to enter its critical section. In this chapter, we consider the problem of assigning priorities to tasks that are blocked on semaphore queues. The ideas in this chapter have not been employed yet in our MDARTS prototype, but they reflect the approach we will take in the future for MDARTS transactions with semaphore queues.

Prior work on real-time multiprocessor synchronization minimizes the global blocking of high-priority tasks at the expense of lower-priority tasks [58, 60]. Global blocking in a multiprocessor system is blocking on semaphores that are shared across processor boundaries. In this chapter, we examine the relationship between global semaphore queue wait times and the schedulability of periodic tasks using rate monotonic scheduling on multiprocessors. We show that in many cases the intuitive assignment of global semaphore queue priorities for tasks actually reduces the overall schedulability of the system. There are two fundamental reasons for this. First, there is no direct correlation between the execution priority of a task and the amount of blocking that it can tolerate and still meet its deadline. Second, higher-frequency tasks contribute proportionately greater blocking delays to lower-frequency remote tasks if the higher-frequency tasks are also given higher semaphore queue priorities. In uniprocessors, blocking for higher-frequency tasks is considered part of the ordinary preemption interval for a task. In multiprocessors, however, blocking for any remote task adds directly to the blocking delay used to determine schedulability. This basic difference between uniprocessor and multiprocessor blocking drastically changes the characteristics of the problem of assigning blocking delays.

For these reasons, our experiments show that a simple FIFO queue for global semaphores usually results in better real-time schedulability on multiprocessors than a priority queue using task execution priorities (henceforth called RMSS, for rate monotonic semaphore scheduling). Furthermore, we show that substantial improvement in schedulability can be achieved if global semaphore queue priorities are explicitly assigned according to the blocking tolerance (the amount of blocking a task can tolerate and still meet its deadline) of a task rather than according to FIFO or RMSS semaphore queue priorities. We also prove that this priority assignment problem is NP-complete and present a heuristic bin packing algorithm that finds a good solution for most task sets.

Early work on scheduling hard real-time systems assumed independence between the tasks to be scheduled [50]. However, most real-time systems require inter-task data sharing that violates the independence assumption of early scheduling algorithms. Furthermore, Mok [53] showed that if tasks make unrestricted use of binary semaphores to enforce mutually exclusive access to shared resources, the problem of determining their schedulability is NP-complete. This is because unrestricted semaphore use can force a high-priority task to wait while a low-priority task holds the lock on a resource. If a medium-priority task that does not use the semaphore then preempts the low-priority task while it is holding the lock, the wait time of the high-priority task can be unbounded. In preemptive scheduling, waiting for a lower-priority task is called *priority inversion*, and the potential for unbounded priority inversion makes the problem of determining schedulability computationally intractable.

However, it is possible to provide mutual exclusion and still guarantee schedulability by restricting semaphore use in various ways. The basic idea is to design protocols that bound the length of time a job might have to wait to acquire a semaphore or access a shared resource. Given such a bound, traditional scheduling strategies such as rate monotonic scheduling or earliest deadline scheduling can be used to guarantee task deadlines [8, 60]. The most common approach to bounding semaphore wait times is to use variants of the priority inheritance protocol (e.g., basic priority inheritance, priority ceiling protocol, semaphore control protocol, kernel priority protocol [59]) to limit wait times due to lower-priority tasks [36, 58, 60, 68]. These protocols temporarily boost the priorities of tasks that are executing critical sections to ensure that they can complete the critical sections within a short, predictable time. This in turn bounds the blocking delays of other tasks that wait for the resources. The priority ceiling protocol and the semaphore control protocol further bound blocking delays and avoid deadlocks by preventing tasks from attempting to acquire semaphores under certain conditions. These "real-time" synchronization protocols were first developed for uniprocessors and then extended to multiprocessors [58, 60, 68].

Some uniprocessor protocols, such as the priority ceiling protocol, do not use explicit semaphore queues. However, real-time multiprocessor synchronization requires queues for the global semaphores. In multiprocessors, the blocking delays also depend on the distribution of tasks that share semaphores across processor boundaries. Since preemptions of critical sections by non-critical section code is prevented or severely constrained by the various priority inheritance protocols, the amount of blocking for a given task and a given semaphore depends only on the length of the critical sections and the periods and semaphore queue priorities of other tasks that share the semaphore. In this chapter, we examine the problem of assigning static global semaphore queue priorities to improve schedulability of real-time multiprocessor applications.

We restrict our analysis to rate monotonic scheduling of periodic tasks composed of sequences of jobs with deadlines corresponding to the task periods (each job J of a task τ must complete its computation within τ's period after its release). A job corresponds to a sequence of instructions that would continuously use the processor until the job finishes if the job were running alone on the processor. Aperiodic tasks can be accommodated within this framework through use of a periodic server [43]. In general, deadline-driven scheduling protocols, which determine execution priorities dynamically, can guarantee higher utilizations than rate monotonic scheduling. However, dynamic priority algorithms are more complex to implement and less stable under overload conditions. For many real-time systems it is better to precompute static priorities and verify system performance off-line rather than spend valuable computation time determining dynamic priorities at runtime. Therefore, we use static priorities both for task execution and for semaphore queue priorities. Furthermore, we assume static assignment of tasks to processors.

The remainder of this chapter is organized as follows. Section 4.2 presents the basic schedulability equations for rate monotonic scheduling and explains why, in multiprocessors, the blocking delays of high-priority tasks should *not* always be minimized. Section 4.3 analyzes the relationship between semaphore queue priority assignment and blocking delays. In Section 4.4, we prove that optimal semaphore queue priority assignment for multiprocessor synchronization is NP-complete. We also present there a heuristic algorithm for solving this problem. Section 4.5 discusses the problem of assigning tasks to processors and relates this problem to that of semaphore queue priority assignment. Section 4.6 describes the experiments we performed to evaluate the different methods for scheduling global semaphore queues. Section 4.7 discusses various implementation issues, and the chapter concludes with Section 4.8.

4.2 Blocking Delays and Schedulability Guarantees

Given rate monotonic scheduling of n periodic tasks with blocking for synchronization, Rajkumar *et al.* [60] proved that satisfaction of the following equation on each processor provides sufficient conditions for schedulability:

$$\forall i, 1 \leq i \leq n \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1) \tag{4.1}$$

In this equation (set of equations, actually), lower-numbered subscripts correspond to higher-priority tasks. C_i, T_i, and B_i are the execution time, period, and blocking time of task τ_i, respectively. The ith equation is a sufficient condition for schedulability of task τ_i. This equation appears very simple, but it warrants careful study. The C_i/T_i components represent the utilization, or fraction of computation time consumed by task τ_i. The number $i(2^{1/i} - 1)$ represents a bound on the utilization of the processor below which task deadlines are guaranteed [50]. As the number of tasks increases, this bound converges to $\ln 2$, or about 70% utilization. This utilization bound provides only a sufficient condition for schedulability; for most task sets, a more complex method called “critical zone analysis” is able to guarantee higher utilizations with rate monotonic scheduling. However, Eq. (4.1) is a useful approximation to critical zone analysis, and it provides insight into the basic properties of rate monotonic scheduling.

The B_i/T_i component represents the effect of blocking on the schedulability of task τ_i. The blocking factor B_i is the amount of time a job J_i may be blocked when it would otherwise be eligible to run. This blocking might result from waiting for a semaphore held by a lower-priority job on the same processor (*local blocking*) or it might result from waiting for a semaphore held by a job of any priority executing on another processor (*remote blocking*). Waiting for higher-priority jobs on the same processor is explicitly excluded from B_i since this is part of the normal preemption time for task τ_i. In other words, the time spent waiting for higher-priority tasks on the same processor is already counted in the C_j/T_j component for $j < i$. Clearly, if the B_i components are not bounded, schedulability cannot be guaranteed. Note that the schedulability equation for each task τ_i depends only on i, the utilizations of tasks with higher-priority than τ_i, and the utilization and blocking of τ_i. In particular, the blocking times of other tasks do not affect the schedulability of a given task. This makes intuitive sense because it is the processor utilizations of other (higher-priority) tasks that reduce the schedulability of a task.

Prior work [60] on real-time synchronization for multiprocessors states: "Another fundamental goal of our synchronization protocol is that whenever possible, we would let a lower-priority job wait for a higher-priority job". This is accomplished for global semaphores by using priority queues to ensure that the highest-priority blocked job will be granted the semaphore next. The justification given in [60] for making lower-priority jobs wait is that the longer periods (T_i) of lower-priority tasks results in less schedulability loss B/T for a given blocking duration B. However, the statement "a given blocking duration B" does not take into account an important characteristic of the problem. In general, the blocking duration B_i associated with a given semaphore queue priority is proportional to the ratio of the periods of the tasks sharing the semaphore: $B_i = K\lceil\frac{T_i}{T_j}\rceil$, where T_j is the period of a task with a higher semaphore queue priority and K is the critical section time for τ_j. If a higher-priority job has a lower semaphore queue priority than a lower-priority job, it waits for at most *one* critical section of the lower-priority job ($\lceil\frac{T_i}{T_j}\rceil = 1$ if $T_i < T_j$). If a lower-priority job also has a lower semaphore queue priority, it may have to wait for *multiple* critical sections of the high-priority job. This factor, on average, increases B_i of the lower-priority task to more than offset the longer period T_i. Therefore, the schedulability loss B_i/T_i is actually greater for the lower-priority task than it would be for the higher-priority task. This is one fundamental reason that it is better to use a FIFO for global semaphores than to use RMSS. We will present a detailed evaluation of this phenomenon in Section 4.6.

Although lower-priority tasks on a uniprocessor are not penalized (made less schedulable) by the blocking of higher-priority tasks, neither do they benefit. This is because the deferred execution of the higher-priority task's critical section must still be completed on the same processor before its deadline. Even if a lower-priority task is granted first use of the shared resource, it can be immediately preempted by the higher-priority task when it exits the critical section. There may be some application-dependent semantic advantage to granting earlier access to a lower-priority task on a uniprocessor (e.g., due to dataflow considerations), but no schedulability advantage is gained by granting the lower-priority task earlier access. The same conclusion can be reached by examination of Eq. (4.1): when a lower-priority task allows a higher-priority task to use a semaphore first, this blocking is considered part of the normal preemption interval of the lower-priority task and does not contribute to that task's B_i component. Alternatively, if the opposite semaphore queue priorities are used, the extra blocking of the higher-priority task *will* add to the higher-priority task's B_i component. Therefore, on uniprocessors, it is always best to assign semaphore

Schedulability equations showing bounds on blocking:	
$U_1 + \frac{B_1}{T_1} \leq 1.00$	$B_1 \leq T_1(1 - U_1)$
$U_1 + U_2 + \frac{B_2}{T_2} \leq .828$	$B_2 \leq T_2(.828 - U_1 - U_2)$
$U_1 + U_2 + U_3 + \frac{B_3}{T_3} \leq .780$	$B_3 \leq T_3(.780 - U_1 - U_2 - U_3)$
$U_1 + U_2 + U_3 + U_4 + \frac{B_4}{T_4} \leq .757$	$B_4 \leq T_4(.757 - U_1 - U_2 - U_3 - U_4)$

Blocking bounds corresponding to possible utilizations:			
$U_1 = .5$	$U_2 = .1$	$U_3 = .05$	$U_4 = .05$
$B_1 \leq .5T_1$	$B_2 \leq .228T_2$	$B_3 \leq .13T_3$	$B_4 \leq .057T_4$
$U_1 = .05$	$U_2 = .05$	$U_3 = .1$	$U_4 = .5$
$B_1 \leq .95T_1$	$B_2 \leq .728T_2$	$B_3 \leq .58T_3$	$B_4 \leq .057T_4$

Table 4.1: Blocking Factor Bounds for Rate Monotonic Scheduling.

queue priorities according to the execution priorities (i.e., RMSS). On multiprocessors, however, the situation is fundamentally different. Higher-priority tasks on remote processors cannot preempt a local task, so the schedulability of local lower-priority tasks can be improved by increasing their global semaphore queue priorities relative to remote tasks.

It is increasingly beneficial to assign remote blocking to higher priority tasks rather than lower-priority tasks as the total number of tasks increases. This is because the additional C/T terms (processor utilization) of the higher-priority tasks further reduce the schedulability and blocking tolerance of the lower-priority tasks. Table 4.1 illustrates the distribution of blocking tolerance for two example task sets. The U_i variables in this table are the C_i/T_i components from Eq. (4.1): utilizations for task τ_i. Although the magnitude of T_i increases as i increases (since we are assuming rate monotonic scheduling), there is a corresponding decrease in the utilization-dependent coefficients. The product of these two components determines the blocking tolerance B of a given task. It is also often better to give higher-priority tasks low semaphore queue priorities if their periods are much shorter than those of lower-priority tasks that share the semaphore (provided, of course, that the higher-priority tasks can tolerate the blocking). This is because a higher-frequency remote task with a higher semaphore queue priority can block a lower-frequency task *multiple* times for each semaphore request of the lower-frequency task.

The following simple example illustrates the greater tolerance of blocking in high-priority tasks. Suppose that we have two jobs J_1 and J_2, both released at time 0, with execution times of 4, and periods 9 and 11, respectively. Furthermore, assume that exactly one of the jobs is blocked for 4 time units when it requests a global semaphore (one held across processor boundaries). If the remote blocking is assigned to the lower-priority job J_2, the schedulability equation for $i = 2$ becomes $.44 + .364 + .364 \leq .83$. This inequality does not hold, so schedulability is not guaranteed. Now assume that the higher-priority job J_1 is blocked instead. The schedulability equations are: for $i = 1$, $.44 + .44 \leq 1$ and for $i = 2$, $.44 + .364 \leq .83$. Both of these inequalities hold, so the tasks are schedulable. Figure 4.1 depicts graphically these two alternatives. In the first case, job J_1 completes its execution before J_2 is allowed to start, and when J_2 is blocked for 4 time units, the delay causes J_2 to miss its deadline of 11. In the second case, when job J_1 is blocked, job J_2 is able to use

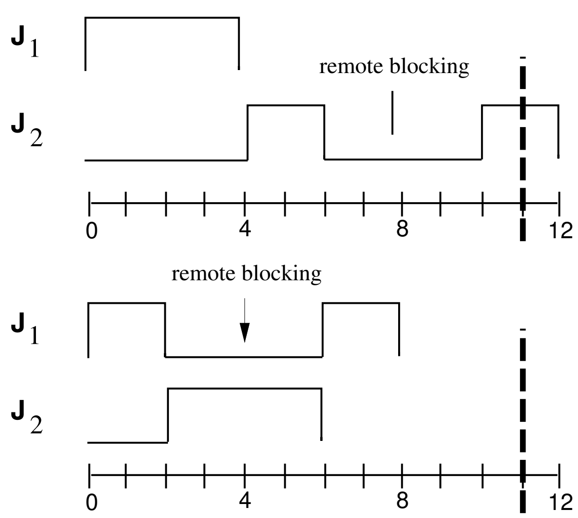


Figure 4.1: Schedulability advantage of blocking high-priority task.

the processor until J_1 is unblocked, and both deadlines are met.

4.3 Priority Queues and Blocking Delays on Multiprocessors

We have seen how rate monotonic scheduling imposes limits on task blocking delays. A full characterization of blocking delays in multiprocessors depends upon the particular scheduling and priority inheritance protocol used. For the purposes of this chapter, we analyze the blocking associated with waiting on a single global semaphore in a multiprocessor. Our analysis applies only to global semaphores; local semaphores should be managed by one of the near-optimal uniprocessor protocols such as the priority ceiling protocol [68]. It is easy to extend our results to derive the total blocking associated with all semaphores. Therefore, our goal is to calculate $B_{i,S}$, the blocking time for job J_i associated with waiting for global semaphore S. To simplify the analysis, we assume that global critical sections are non-preemptible, which approximates the behavior of the modified priority ceiling protocol proposed for multiprocessor synchronization [60]. We define the following notation. Note that J_i might contain multiple critical sections guarded by S. Furthermore, unlike Eq. (4.1), the task numbers in our notation do not correlate with priorities. On a multiprocessor, task 53 might be the highest-priority task on one of the processors, so the notation of Eq. (4.1) is inadequate for this domain.

$\wp_j$: The processor to which J_i is assigned.

T_i: The period of task τ_i.

$P_{k,S}$: The semaphore queue priority for job J_k when it waits for S. As discussed thus far, this priority is independent of the execution priority of J_k.

$\{JL_{i,j,S}\}$: The set of local jobs on $\wp_j$ that use S and have lower execution priority than J_i.

$\{J_{r,S}\}$: The set of jobs assigned to processor $\wp_r \neq \wp_j$ that use semaphore S.

$\{JHQP_{i,S}\}$: The set of jobs $J_k \in \{JL_{i,j,S}\} \cup \{J_{r,S}\}$ with $P_{k,S} > P_{i,S}$.

$\{JLQP_{i,S}\}$: The set of jobs $J_k \in \{JL_{i,j,S}\} \cup \{J_{r,S}\}$ with $P_{k,S} < P_{i,S}$.

$CS_{i,S}$: The maximum time required by job J_i to execute a critical section guarded by S.

$CS_{lmax,i,S}$: The maximum critical section time for S of jobs $J_k \in \{JLQP_{i,S}\}$.

$NC_{i,S}$: The number of times J_i enters a critical section guarded by S.

$LNUM$: $min(NC_{i,S}\ ,\ \sum_k (NC_{k,S} \times \lceil \frac{T_i}{T_k} \rceil)$ for $J_k \in \{JLQP_{i,S}\})$.

Given these definitions, the blocking $B_{i,S}$ is bounded by:

$$B_{i,S} \leq (LNUM \times CS_{lmax,i,S}) + \sum_k (NC_{k,S} \times CS_{k,S} \times \lceil \frac{T_i}{T_k} \rceil) \text{ for } J_k \in \{JHQP_{i,S}\})$$

At most $NC_{k,S} \times \lceil \frac{T_i}{T_k} \rceil$ critical sections guarded by S for each J_k can block J_i since the number of instances of J_k within τ_i's period is bounded by $\lceil \frac{T_i}{T_k} \rceil$. Critical zone analysis cannot reduce this bound, because the tasks in $\{JHQP_{i,S}\}$ with higher frequency than τ_i are on remote processors. The extra blocking represented by $LNUM \times CS_{lmax,i,S}$ accounts for the possibility that a task (local or remote) with a lower semaphore queue priority will already be using the semaphore when task τ_i attempts to access it. The number of jobs $J_k \in \{JHQP_{i,S}\}$ depends on the task distribution across the multiprocessor and the semaphore queue priority assigned to J_i for semaphore S.

If we take the task distribution in the multiprocessor as given (we address the issue of task allocation in Section 4.5), the blocking associated with semaphore S for J_i is primarily determined by the semaphore queue priorities. Therefore, it is possible to manipulate the distribution of blocking associated with S across the different tasks that use S by assigning the semaphore queue priorities $P_{k,S}$. Semaphore queue priority assignment provides an added degree of freedom to the design of a synchronization strategy for multiprocessor scheduling. By allocating blocking delays in this way, it is possible to improve the overall schedulability of the system.

Now let us consider the RMSS semaphore queue priority assignment proposed in [58, 60]. As we have seen, lower-priority tasks can be less tolerant of blocking delays than higher-priority tasks. If task execution priorities are used for semaphore queues, as much blocking as possible is assigned to the lower-priority tasks. In some cases, this will lead to a violation of the blocking bound of a lower-priority task, i.e., the task will not be schedulable. If the semaphore queue priorities cannot be adjusted, the only recourse is to somehow modify the task utilizations $\frac{C_i}{T_i}$ or change the task distribution on the multiprocessor. Neither of these alternatives is always feasible. The periods of tasks are often fixed according to physical characteristics of the real-time system, and the computation times C_i cannot always be reduced without compromising the quality of the solution. Moving the task to a different processor will not necessarily make the task schedulable, and it may cause other tasks on the destination processor to become unschedulable. Fortunately, it is often relatively easy to improve the schedulability of the system by taking a more flexible approach to global semaphore queue priority assignment.

The previous discussion might lead one to conclude that priorities in global semaphore queues should be the inverse of the task execution priorities. However, this is not always true either. A high-priority task may have such a short period that it cannot afford to wait long on its semaphore queues. In general, there is no fixed relationship between blocking tolerance and task execution priorities. It is necessary to conduct a search over possible semaphore queue priority assignments for each particular task set to find assignments that will guarantee schedulability.

4.4 The Semaphore Queue Priority Assignment Problem

Given a distribution of tasks sharing resources on a multiprocessor such that all tasks are schedulable via rate monotonic scheduling without global semaphore blocking delays, we would like to know whether the tasks can be assigned global semaphore queue priorities such that they are still schedulable. Furthermore, we would like to have an efficient algorithm to generate a solution to the priority assignment problem. We call this problem SQPA-RMS. Unfortunately, this problem is computationally intractable.

Theorem 1 *The semaphore queue priority assignment problem for rate monotonic scheduling (SQPA-RMS) is NP-complete.*

Proof: To show that SQPA-RMS is in NP, for an instance of the problem, we let the set of priority assignments $\{P_{i,S}\}$ be the certificate (where $P_{i,S}$ is the semaphore queue priority for task τ_i and semaphore S). Checking whether task deadlines are guaranteed can be performed in polynomial time by calculating the blocking factors for each task τ_i using $\{P_{i,S}\}$, as described in Section 4.3, and applying Eq. (4.1) or critical zone analysis.

We now show that SQPA-RMS is NP-hard by reducing PARTITION [24] to an instance of SQPA-RMS.

Suppose that we have a multiprocessor with 3 processors. Processor $\wp_1$ will be assigned a task that uses all n global semaphores but has a low utilization so that it can always tolerate the lowest semaphore queue priority (priority 1). Processor $\wp_1$ we call the "blocking processor" since it serves only to supply blocking delays to the system. Without this processor, the semaphore queue priority assignment of tasks on the two "partition processors" would make no difference, since the queue could never be longer than one. The other two processors are each assigned a single task τ_i. Each of these tasks has the same execution time C and the same period T, and each task τ_i will use all n global semaphores.

Now for a given instance of PARTITION, we define n global semaphores where $n = |A|$ and map each of the n items $a \in A$ to the decision to assign queue priority 3 or queue priority 2 for a given semaphore and a given task τ_i on one of the partition processors. Furthermore, we set the critical section times for each semaphore such that the PARTITION size function $s(a) \in Z^+$ corresponds to the difference in blocking between having priority 3 or priority 2 in the semaphore queue. Finally, we adjust the utilization and period of the two tasks τ_i such that the available blocking B for each is exactly $B_{min} + 1/2 \sum_{a \in A} s(a)$. B_{min} is the best-case blocking for the partition processors, corresponding to having priority 3 for all semaphores. Now we have a symmetric problem where each processor can tolerate exactly $1/2$ of the total blocking associated with having the lowest semaphore queue priorities. All of these transformation steps can be performed in polynomial time.

Suppose that a subset A' exists with $\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a)$. Then if we assign semaphore queue priority 3 to τ_2 for semaphores associated with $a \in A'$ and priority 2 for the rest (and vice versa for τ_3 on processor $\wp_3$), the total blocking B for each task τ_i will be exactly $B_{min} + 1/2 \sum_{a \in A} s(a)$, which will not exceed the blocking tolerance. Hence, the task set will be schedulable. Conversely, assume that we have determined a priority assignment for each of the n semaphores and each task τ_i such that the blocking tolerance B is not violated. Then if we choose the items a that correspond to the priority assignment 3 for τ_2, we will have constructed a subset A' of A with $\sum_{a \in A'} s(a) = \sum_{a \in A-A'} s(a)$.

We have shown how to reduce PARTITION to an instance of SQPA-RMS, so SQPA-RMS is NP-hard. Since SQPA-RMS is in NP and is NP-hard, it is NP-complete. □

4.4.1 The BINP algorithm

Unless P=NP, no efficient algorithm solves SQPA-RMS. However, we have developed an efficient algorithm that performs well for most task sets. Our approach is essentially a heuristic bin packing algorithm. Before presenting the algorithm, which we call BINP, we review the characteristics of the problem that guided our selection of heuristics. Each task in the multiprocessor has a finite blocking tolerance defined by Eq. (4.1) and illustrated in Table 4.1. This blocking tolerance can be considered the "bin size" of that task. Our goal is to assign semaphore queue priorities to each task such that the resultant blocking does not exceed the blocking tolerance of any task.

Whenever a lower global semaphore queue priority is assigned to a task, other tasks sharing that semaphore benefit. From the perspective of tasks on one processor, it is always preferable that tasks on another processor do more waiting. When possible, it is preferable that tasks with short periods (high execution priorities) be assigned a low semaphore priority $P_{i,S}$ since high frequency tasks add proportionally more blocking to the tasks with lower semaphore priorities and incur proportionally less blocking from the tasks with higher semaphore priorities. This is because global blocking is a function of the ratio of task periods. If a task has only one source of blocking (one semaphore) not yet assigned, the maximum allowable blocking should be chosen for that task, because any excess task blocking capacity is effectively wasted once all of its semaphore priorities are chosen.

With these characteristics in mind, we present our heuristic algorithm. Our basic strategy is to use a modified ordered best fit bin packing algorithm, assigning the largest blocking elements (lowest semaphore queue priorities for the most heavily used semaphores) first to those bins (tasks) that have the most blocking capacity. Because not all tasks use every semaphore, we really have several bin packing problems that are overlaid on overlapping subsets of bins. The asymmetries of the problem make analysis difficult, but the best fit with largest items first heuristic performs well in practice.

1. Calculate the blocking bounds for each task from Eq. (4.1) or from the more accurate critical zone analysis.

2. Determine the blocking delays associated with local semaphores using whatever protocol is chosen to manage them (e.g., priority ceiling protocol). Subtract these delays from the blocking bounds determined in Step 1. The result is the available absolute blocking tolerance for each task. If any of these are negative, report failure immediately.

3. Identify the semaphore S with the largest unassigned blocking delay. An approximate method for determining this is: for each semaphore with priorities still unassigned for some set of tasks $\{\tau\}$, compute $\sum_{\tau_k \in \{\tau\}} T_{maxp} \times NC_{k,S}/T_k$ where T_{maxp} is the maximum period of the tasks in $\{\tau\}$. Choose the semaphore with the largest sum.

4. Find the task τ that uses semaphore S and that has the largest measure of blocking tolerance. This can be defined as the largest absolute blocking tolerance or the largest

ratio of blocking tolerance to unassigned semaphores (excluding S) for that task. If one or more tasks has enough absolute blocking tolerance for the blocking currently being assigned and if it has no other unassigned semaphores, then choose the task with the highest execution priority (shortest period) in this group. This rule is an elaboration of the selection method that uses the largest ratio of blocking tolerance to unassigned semaphores.

5. Assign the lowest unassigned semaphore queue priority for S to task τ. As semaphore priorities are assigned to a task, the absolute blocking tolerance of that task is reduced to reflect the blocking delay associated with that semaphore priority assignment.

6. Repeat the previous three steps until all semaphore queue priorities are assigned for all tasks.

7. Verify the schedulability of the task set using Eq. (4.1) or critical zone analysis.

4.4.2 Complexity analysis

If there are k semaphores, each shared by an average of T_{ave} tasks and by at most T_{max} tasks, there are a total of $k \times T_{ave}$ priority assignments to make. For each priority assignment, BINP iterates over the unassigned tasks for that semaphore, which number at most T_{max}. When a priority assignment is made, the unassigned blocking delay for that semaphore is adjusted, and the semaphore is inserted into a priority heap. This insert operation has complexity $\lg k$. The overall complexity of BINP is thus: $O(kT_{ave}(\lg k + T_{max}))$. If T_{max} is bounded, the algorithm is essentially $O(k \lg k)$. Otherwise, it is quadratic in T_{max}. In either case, this is a relatively efficient algorithm.

Now, consider the priority assignment method of [58, 60]: assign lowest semaphore queue priorities to the lowest-priority tasks. This is a constant time algorithm, but it is essentially a bin packing strategy in which the largest k items are assigned to a predetermined set of bins, which could be the smallest bins. If this priority assignment overflows a bin's capacity, the algorithm returns failure. Clearly, the bin packing algorithm we propose should perform much better than the method used in [58, 60]. By performing better, we mean that given some population of task sets, more will be schedulable with our algorithm. To verify this claim, we have implemented our algorithm and conducted extensive experiments comparing our approach with the previous approach and with simple FIFO queues. The results of our experiments are presented in Section 4.6. We also examine implementation issues associated with the different algorithms in Section 4.7.

4.5 The Task Allocation Problem

In the discussion so far, we have assumed that the task distribution across the multiprocessor was fixed and the problem was to choose the semaphore queue priorities. In reality, one often needs to solve both problems: first task allocation and then semaphore queue priority assignment. Since the problem of allocating tasks to processors on a multiprocessor is known to be NP-complete [24] even when no resource sharing (other than processors) is considered, there is no computationally efficient solution for this problem (unless P=NP). The potential for blocking on semaphore queues adds another level of complexity to an already very difficult problem. A task allocation algorithm should consider the impact of resource sharing since this will affect the overall schedulability of the system. However, the resource sharing costs depend in part on the task allocation. Therefore, if a task allocation algorithm conducts a search over possible task allocations, the resource sharing costs have to be computed for each point in the search space. The semaphore queue priority assignment method we propose requires more computation than the fixed priority methods of FIFO and RMSS (i.e., $O(kT_{ave}(\lg k + T_{max}))$ rather than constant time). Our algorithm is relatively efficient, but if it is employed during task allocation, some additional overhead is incurred compared to FIFO and RMSS. However, the solution generated by BINP, as shown in Section 4.6, will usually be superior to that of FIFO or RMSS.

It is important to remember that this overhead is paid off-line, before any real-time processing occurs. Furthermore, it is not absolutely necessary to use our algorithm at each point in a task allocation search. It would be perfectly valid to use FIFO or RMSS priorities to arrive at a task allocation and then use BINP to improve the schedulability for that allocation. This might result in a suboptimal combination of task allocation and semaphore queue priorities, but we cannot efficiently find an optimal combination in the first place, since task allocation itself is NP-complete. Regardless of the method used for task allocation, once that allocation is determined, our method for semaphore queue priority assignment should lead to better schedulability than either FIFO or RMSS for most task sets. This claim is supported by our experiments with randomly-generated task sets and by the intuitive argument that when placing objects into bins it is better to consider the relative sizes of the objects and bins than to follow some fixed assignment policy.

4.6 Experiments

To illustrate the advantages of our algorithm over RMSS and simple FIFO queues, we implemented all three methods and compared their performance in scheduling a large number of randomly-generated task sets. Our task-set generator program took the following parameters: target utilization for the processors, number of task sets to generate, number of processors, average number of tasks per processor, number of semaphores, and a flag to vary or keep constant the critical section times for each task using a semaphore. First, we established a range of periods for the tasks from 100 to 3000. Second, we chose the nominal critical section time for each semaphore from a uniform distribution between 0.1 and 0.5 of the average expected computation time of a task (average period $\times$ target utilization $\div$ average number of tasks per processor). Next, we began generating tasks for each processor until the assigned utilization reached the target utilization.

For each task, we first randomly chose a utilization from a uniform distribution between one third and twice the average utilization (target utilization for each processor $\div$ average number of tasks). If the chosen utilization plus that of the tasks already assigned to that processor exceeded the target utilization bound, we reset the chosen utilization to equal the difference. In this way, we ensured that every processor would have the same utilization load. In a realistic system, not every processor is equally loaded, but we were interested in examining the behavior of the system when the blocking delays were close to violating the schedulability of the processor. By loading all processors equally, we were able to move all of the processors near this region of marginal schedulability in a consistent manner. Equal loading also diminished somewhat the expected advantage of our semaphore assignment strategy since BINP is able to add extra blocking to tasks on lightly-loaded processors. Nevertheless, in the interest of simplicity, we used uniform utilizations. Given the task's chosen utilization, we chose the task's period from a uniform distribution between 100 and

3000 and derived the corresponding computation time. We also set the priority for the task according to the rate monotonic scheduling discipline. Once each task's execution parameters were set, we used the following method to choose its semaphores.

For each task, we randomly chose a fraction of its computation time to devote to executing global critical sections. This is an important parameter because it partially determines how many semaphores will be used by the task and thus how much blocking will be incurred. Therefore, we decided to select this parameter randomly rather than choose some fixed value for it. The range we chose was between 0.2 and 0.8 of the computation time, so on average about half of a task's computation time will be spent in critical sections. Real applications might spend less time executing global critical sections, but we chose this range to examine the schedulability characteristics of the three semaphore queue priority assignment methods under consideration. Obviously, if global semaphore critical sections represent only a very small fraction of the computation in a given application, their impact on schedulability will also be small (if priority inversion is limited).

We next examined the flag for varying critical section times. If the flag was true, we randomly chose an additional scaling factor between 0.25 and 1.75 that was multiplied by the semaphore's nominal critical section time to determine the critical section time for that task and that semaphore. We chose a different critical section scaling factor for each semaphore used by each task. To choose the semaphores, we randomly selected a semaphore from the semaphore set and checked whether adding its critical section time would exceed the fraction of computation time bound for critical sections for that task. If this bound was not exceeded, we assigned that semaphore to the task. If the same semaphore was chosen more than once, we incremented the number of times the semaphore was used by each job of the task ($NC_{k,S}$). If the bound was exceeded, we skipped that semaphore and chose another. If five selections in a row exceeded the bound, we exited the semaphore assignment loop for that task. Because of this termination condition, often the sum of critical sections assigned to a task did not quite reach the fraction of execution time bound.

4.6.1 Description of task sets

For our primary experiments, we generated 5,400 different task sets. These task sets were generated in groups of 50 sets for each combination of [3, 6, or 10 processors], [3, 6 or 10 tasks per processor], [5, 10, or 20 global semaphores], [processor utilizations of 0.6 or 0.7], and [constant or varying critical section times for semaphores]. For our schedulability analysis, we used critical zone analysis rather than Eq. (4.1) because it is a more accurate method for determining schedulability. If Eq. (4.1) is used, the blocking bounds are slightly tighter, which makes the task sets more difficult to schedule. The relative performance of the semaphore scheduling methods remains the same regardless of which method is used.

We first checked how many of the task sets each method successfully scheduled. Table 4.2 presents this information. Not surprisingly, more task sets were schedulable with lower utilization and with constant critical section times (constant only for a given semaphore, different semaphores were assigned different critical section times). Processors with lower utilizations have larger bounds on blocking for the lowest-priority tasks, and it is easier to schedule resources if they are of uniform size.

The data show a clear trend with BINP performing much better than FIFO, which in turn performs much better than RMSS. We also examined which combination of processor number and number of tasks corresponded to the schedulable and unschedulable task sets. The most significant factor was the number of processors. Of the 2,721 task sets scheduled

critical section times	utilization	BINP	FIFO	RMSS
constant	0.6	987	522	275
varied	0.6	748	411	206
constant	0.7	602	292	109
varied	0.7	384	187	64
total		2721	1412	654

Table 4.2: Task Sets Scheduled by Each Method (Each Row Corresponds to a Different Group of 1,350 Task Sets).

by BINP, 59% were from the 3 processor task sets (1/3 of the overall task sets had 3 processors). For FIFO queues, 84% of the 1,412 schedulable task sets had 3 processors; for RMSS, 95.7% of the 654 schedulable sets had 3 processors. It was easier to schedule fewer processors in our test sets since the number of semaphores was independent of the number of processors. Fewer processors meant fewer total tasks in the system and hence less contention for the fixed number of semaphores. We ran some additional experiments with 100 processors to test this hypothesis. With 200 semaphores, BINP was able to consistently schedule task sets with 100 processors, 6 tasks per processor, 0.6 utilization, and constant critical section times. The FIFO and RMSS methods could not schedule any of these task sets. However, if there were only 100 or 50 semaphores, none of the priority assignment methods could schedule any of the 100 processor task sets.

It is also likely that with more processors there is a greater chance that one of the processors has a task/semaphore distribution that is especially difficult to schedule. This would make the multiprocessor less likely to be schedulable than an individual processor. To investigate this possibility, we counted how many individual processors were schedulable with each method for our test sets. If entire task sets are considered, as in Table 4.2, BINP scheduled 50% of the task sets, FIFO scheduled 26%, and RMSS scheduled 12%. For individual processors within the multiprocessor, BINP scheduled 47.9%, FIFO scheduled 29.5%, and RMSS scheduled 19.9%. These numbers confirm our suspicion that the FIFO and RMSS methods sometimes are unable to schedule a multiprocessor due to unfavorable task distributions on individual processors. However, BINP is able to modify its blocking assignments to compensate for processors that are more difficult to schedule. Therefore, in our task sets, BINP was able to schedule approximately the same percentage of complete task sets as individual processors.

We next investigated how the different methods compared when individual task sets were considered. One might think that the different semaphore priority assignment methods would be suited to different task sets. For example, are some of the task sets schedulable via RMSS but not BINP, and vice versa? Table 4.3 addresses this question. The answer is "no," RMSS could not schedule any of the task sets that BINP could not schedule. On the other hand, BINP scheduled 2067 task sets that RMSS could not. Likewise, BINP dominated FIFO and FIFO dominated RMSS in terms of scheduling individual task sets. Thus, the performance differences between these methods are significant on average, and they also hold for individual cases.

Since the three methods have significant differences in ability to schedule task sets,

Method	BINP	FIFO	RMSS
BINP	-	7	0
FIFO	1316	-	15
RMSS	2067	773	-

Table 4.3: Individual Task Sets Schedulable by Method of Column but not by Method of Row.

it is important to measure more than simply the number of task sets scheduled. This is because it is possible to make the best method look arbitrarily good by adjusting the parameters that determine how difficult the task sets are to schedule. For example, BINP was able to schedule 50% of our sample task sets. By reducing the number of very difficult task sets (those with the most contention for semaphores) and increasing the number of moderately difficult task sets (those that BINP can usually schedule but the other methods cannot), we could disproportionately increase the percentage of task sets schedulable by BINP. To eliminate this source of bias and to better quantify the relative performances of the different methods, we investigated how close the unschedulable task sets were to being schedulable under each method. To answer this question, we kept the semaphore queue priorities the same and gradually decreased the utilization of all processors in the multiprocessor until the task set became schedulable. We did this by leaving most task set values unchanged (periods, semaphore priorities, etc.) while decreasing the computation time of the tasks and of the critical sections. The percentage by which it is necessary to reduce the utilization to achieve schedulability we call "*delta*." Task sets that were close to being schedulable typically became schedulable with a *delta* of five or ten. Task sets that are far from schedulable might have a *delta* of 40 or 50. A *delta* of 30 with an initial utilization of 0.6 means the utilization must be reduced to 0.42 before the task set becomes schedulable.

To determine *delta*, we scaled back the computation times uniformly across all of the processors until the task set became schedulable. This led to some unnecessary reduction of utilization for individual processors that were already schedulable, but it simplified the metric. In realistic systems, it may not be possible to reduce computation times significantly without reducing the quality of the solution. Usually the physical characteristics of the real-time system determine the periods, so it is even less realistic to reduce utilization by extending task periods. A uniform reduction in computation time is equivalent to using a faster computer. Realistic or not, *delta* provides a useful metric to determine how close a task set is to being schedulable by a given semaphore queue priority method. We determined two *delta* factors for BINP: the *delta* when the semaphore priorities were kept constant as the utilization changed, and the *delta* when the semaphore priorities were recomputed as the utilization changed. We call the latter the "Reassign BINP" *delta*.

Table 4.4 shows the average *delta* values for the task sets that were unschedulable by BINP. The row labeled "Most difficult" averaged the *delta*s of the groups of task sets for which BINP could not schedule any of the 50 similar task sets (recall that we generated 50 task sets for each combination of task parameters such as number of processors). These task sets were the most difficult to schedule, and Table 4.4 shows that they had the largest

	Test sets	Reassign BINP	BINP	FIFO	RMSS
Most difficult	1350	26.5	38.3	56.8	64.3
Moderately difficult	1329	9.4	12	32.9	44.9
Overall	2679	18	25.4	44.9	54.7

Table 4.4: Average Percentage *delta*s for Unschedulable Task Sets, a Smaller *delta* Means Closer to Being Schedulable.

Method	BINP	FIFO	RMSS
BINP	-	48	7
FIFO	3922	-	438
RMSS	4732	4252	-

Table 4.5: Individual Task Sets for Which the Method of the Column Performed Better than the Method of the Row.

*delta*s. The "Moderately difficult" row in Table 4.4 averages the *delta*s of task sets for which some of the 50 similar sets were schedulable and some were not. The "Overall" row averages all of the cases from the other two rows. The *delta* factors provide a more useful characterization of the relative performances of the three priority assignment methods than simply how many task sets are schedulable. As Table 4.4 shows, on average, bin packing could schedule about 20% more utilization than FIFO ($44.9 - 25.4$), which in turn could schedule about 10% more utilization than RMSS ($54.7 - 44.9$). The differences in utilization schedulable by each method depends heavily on the fraction of computation time spent in global critical sections. The percentages reported here are with respect to our task sets. Task sets for which less time is spent in critical sections would probably have lower *delta* percentages but the same pattern of relative performance.

Reassigning the semaphore queue priorities as the utilization was scaled back helped substantially for those test cases that were difficult to schedule (the "Most difficult" cases) but did not help much for test cases that were originally nearly schedulable (the "Moderately difficult" cases). This is because the shape of the blocking bound curve changes as computation times are reduced. The more the computation times are scaled back the more the shape changes and the more opportunity for improvement through changing the blocking distribution to reflect the new bin sizes.

We next investigated how the different methods compared when the *delta*s of individual task sets were examined. This information is summarized in Table 4.5. Table 4.5 expands upon Table 4.3 by counting all of the individual task sets for which one method performed better than another. Performing better is defined as either successfully scheduling the task set when the other method could not or as having a smaller *delta* than the other method. These results are consistent with that reported in Table 4.3: By a wide margin, BINP performs better than FIFO and RMSS; FIFO performs better than RMSS by a lesser

```
run 8 0.7 util 3 cpus 6 tasks 5 sems
#nominal semaphore CS times
45 32 70 46 63
#task cpu priority period ctime ; sem# NCS CSscale ...
   1  0 273 1095 66 ;0 1 0.62
   2  0 271 1106 81
   3  0 193 1553 290 ;0 2 0.48; 3 1 1.7
   4  0 152 1966 144 ;3 1 0.9
   5  0 150 1989 127 ;1 1 1.4; 4 1 0.6
   6  0 129 2315 424 ;0 1 0.28; 2 2 0.74; 3 2 1.5; 4 1 1.2
   7  0 122 2453 147 ;0 1 0.57; 3 1 0.3; 4 1 0.81
   8  1 395 758 108 ;1 3 0.33
   9  1 394 760 115 ;0 1 0.85; 1 1 0.35; 2 1 0.36
   10 1 131 2284 333 ;0 1 0.79; 1 2 1.7
   11 1 117 2556 293 ;0 3 0.29; 3 4 0.31; 4 1 1.3
   12 1 104 2874 419 ;0 1 0.62; 1 1 0.77; 3 1 1.4
   13 2 622 482 45
   14 2 437 686 27
   15 2 193 1547 235 ;3 1 1
   16 2 115 2603 365 ;0 1 1.6; 3 3 0.3; 4 1 0.62
   17 2 110 2722 244 ;0 2 0.9; 1 1 0.9; 2 1 0.92
   18 2 108 2764 513 ;0 1 1.5; 2 2 1.4
```

Figure 4.2: Example task set.

margin. To keep the comparison fair, the *delta* of BINP in Table 4.5 is the one that kept the original semaphore queue priorities (it is not the "Reassign BINP" *delta*).

4.6.2 A specific task set

Now that we have characterized our entire task sets, we will examine a particular case in detail to gain insight into the behaviors of the different semaphore queue priority assignment methods. Figure 4.2 shows the task set (to make the listing easier to correlate with the graphs, the tasks in Figure 4.2 were renumbered and sorted in decreasing priority order for each processor). This particular task set was one of the 50 task sets generated with 0.7 utilization, 3 processors, average of 6 tasks per processor, 5 semaphores, and varying critical section times. For these parameters, BINP was able to schedule 16 of the 50 task sets. The average *delta* factors for the 34 unschedulable task sets were: "Reassign BINP"= 8.9 BINP= 9.7 FIFO= 24.1 RMSS= 32.2. For the particular task set considered here, the *delta* factors were: "Reassign BINP"= 8 BINP= 10 FIFO= 23 RMSS= 31. Therefore, this task set is fairly representative of the group of 34 unschedulable ones with the same task generation parameters.

Processor 0 has 7 tasks, while processor 1 has only five. This is because we randomize the utilizations for tasks as we generate them and stop when the processor utilization limit is reached, not when a certain number of tasks are chosen. Likewise, three of the tasks were not assigned semaphores. This is because their computation times were small, and the method we used to select semaphores failed to assign a semaphore after five random selections.

Figure 4.3 shows the blocking distributions and bounds for the tasks when BINP is used to assign the semaphore queue priorities. In these graphs, the bars represent worst-case blocking for each task associated with its chosen semaphore queue priorities. The line shows the bound on blocking below which the task is schedulable with rate monotonic

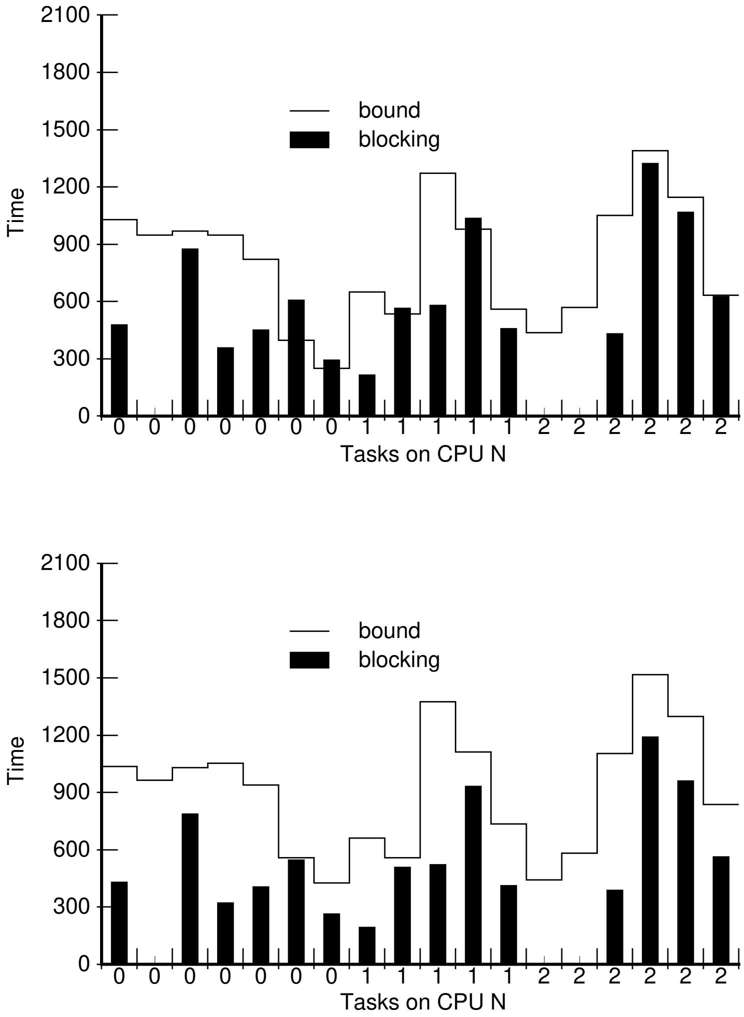


Figure 4.3: Bin packing blocking before and after a 10 percent reduction in utilization.

scheduling. The tasks on each processor are shown in decreasing priority order from left to right on the X axis, and the processor numbers for the tasks are shown below each task's bar. The lower graph in Figure 4.3 shows that when utilizations are reduced by 10% (*delta* = 10), the task set becomes schedulable with the original BINP semaphore queue priorities. If one examines the blocking bounds for each processor, it is apparent that there is no direct relationship between task priority and blocking bound. However, the highest-priority (leftmost) and lowest-priority (rightmost) tasks on each processor tend to have the tightest blocking bounds. The highest-priority tasks have tight blocking bounds because they have short periods. The lowest-priority tasks have tight bounds because of their lower execution priority. As the utilizations are scaled back, the blocking times (semaphore critical section times) decrease and the bounds increase until the blocking no longer exceeds the bound. At that point, all of the (modified) tasks are guaranteed to be schedulable.

Figure 4.4 shows the blocking distributions and bounds for the tasks when FIFO is used for the semaphore queues. If the top graphs for BINP and FIFO are compared, one

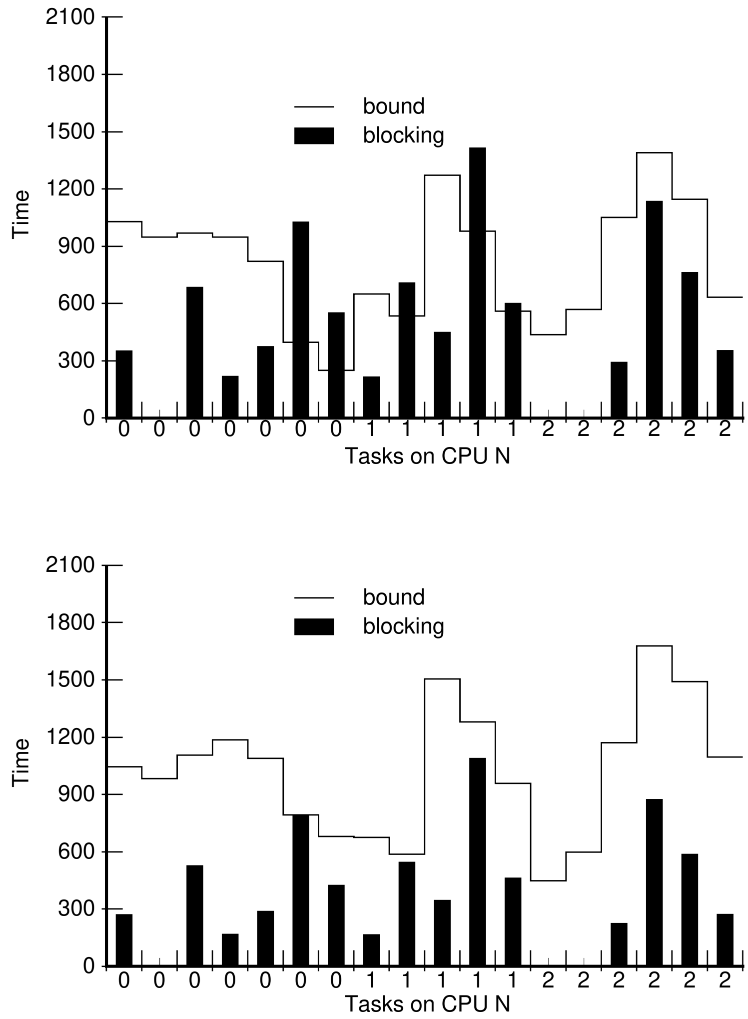


Figure 4.4: FIFO blocking before and after a 23 percent reduction in utilization.

can see that FIFO assigns less blocking to the higher-priority tasks on each processor than BINP does. With FIFO, the lower-priority tasks thus exceeded their bounds by a greater percentage than with BINP. Therefore, the original utilization of 0.7 had to be scaled back by 23% before FIFO was able to schedule the task set. The blocking bound curve for the tasks with scaled-back utilization differs somewhat in shape from the original bound curve since the increase in bound for each task is the product of the task periods and the reduction in higher-priority utilization (recall Eq. (4.1) and Table 4.1). Although we used critical zone analysis to determine the bounds for each testset, Eq. (4.1) approximates it. Thus, the blocking bound increases disproportionately for lower-priority tasks, which have longer periods and a larger sum of higher-priority task utilizations.

Figure 4.5 shows the blocking distributions and bounds for the tasks when RMSS is used for the semaphore queues. RMSS assigned even more blocking to the low-priority tasks than FIFO and thereby severely exceeded the blocking bounds of the lowest-priority tasks. A *delta* of 31 was required to make the tasks schedulable with RMSS. In general,

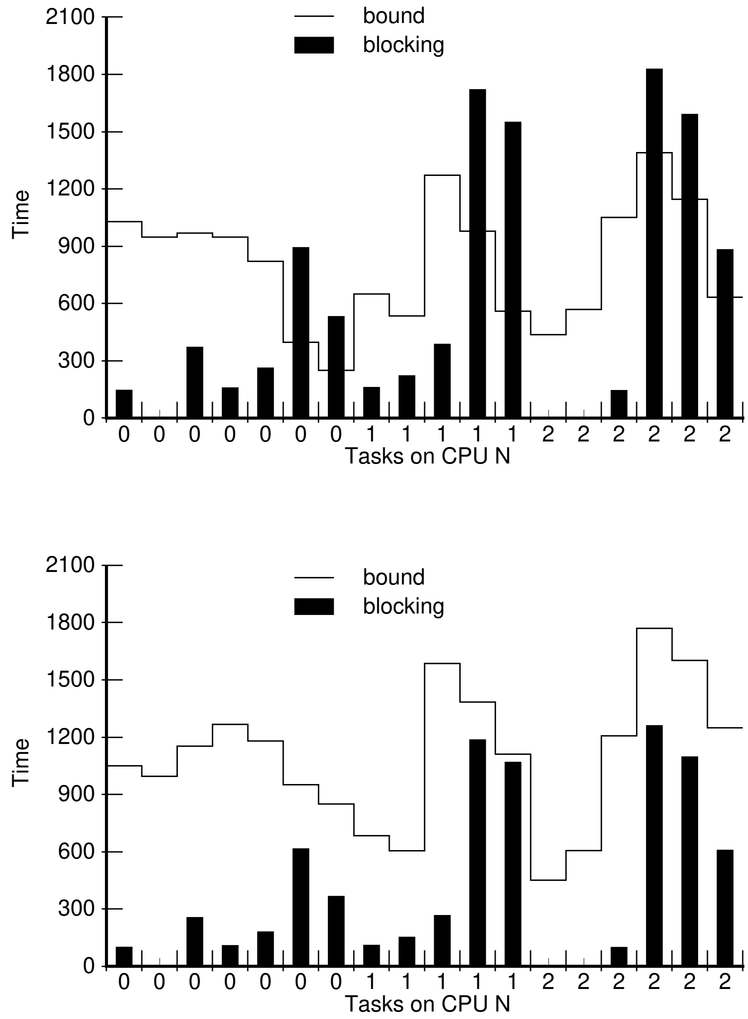


Figure 4.5: RMSS blocking before and after a 31 percent reduction in utilization.

the inflexibility of FIFO and RMSS leads to assigning too much blocking to tasks with low blocking tolerance, thereby making them unschedulable. RMSS assigns as much blocking as possible to the lowest priority tasks, and performs even worse than FIFO for most of our task sets.

4.7 Implementation Issues

We have shown that BINP has significant advantages over both FIFO and RMSS, but it also is important to consider the relative complexity and overhead associated with implementing the different methods. Clearly, BINP is more complex to implement than either FIFO or RMSS. FIFO and RMSS are constant-time algorithms for which the priorities are immediately known. Our BINP algorithm runs in $O(kT_{ave}(\lg k + T_{max}))$ time for k semaphores, each shared by on average T_{ave} and at most T_{max} tasks. A more efficient bin packing algorithm for this problem may exist, but it would still be slower than FIFO or

RMSS. Although BINP is more complex to implement, the performance advantages could be substantial for real-time multiprocessor applications with heavy semaphore use. Furthermore, the computation of BINP is performed off-line, so it does not add extra overhead at runtime. Real-time application designers usually perform off-line schedulability analysis and task allocation, so running BINP would simply be an extra part of that activity.

BINP does not require any significant runtime overhead compared with RMSS, with the possible exception of an additional table for storing the semaphore queue priorities (the RMSS priority can be inferred from the task priority and thus need not be stored separately). Nevertheless, for applications that do not use global semaphores heavily it might be preferable to simply use FIFO priorities. In our experiments, FIFO was almost always superior to RMSS, so FIFO is probably the best alternative to BINP. For some applications, in which the high-priority tasks have particularly tight blocking bounds, RMSS might be better than FIFO. However, in most cases either FIFO or BINP should be used.

Both BINP and RMSS require a priority queue for semaphores. The extra overhead of maintaining a priority queue rather than a FIFO must be justified by the superior performance of BINP (but not RMSS, which performs worse than a FIFO in our testsets). Since suspending a task on a semaphore generally requires substantial overhead in the first place, maintaining a priority queue rather than a FIFO should not add significantly to the overhead. Ultimately, the question of whether a priority queue is justified depends upon the application and the implementation of the queues. The operating system should probably provide both types of queues unless the overhead differences are trivial, in which case only priority queues should be provided (since a priority queue can emulate a FIFO).

We believe that the determination of the semaphore queue priority should be the responsibility of the application rather than the operating system. This allows an application programmer to explicitly control the relative blocking times of tasks rather than commit to a particular semaphore scheduling policy. The operating system could use a FIFO by default and allow applications to specify priorities for those global semaphore queues that require more control.

If tasks must be dynamically added or removed from the multiprocessor, it is possible to use BINP to precompute semaphore queue priorities corresponding to different task sets on the system (different modes). Real-time systems tend to be much more consistent in their task sets than conventional multiprocessing systems, so the different modes are likely to be known in advance. Alternatively, suppose an unexpected task needs to be added to a running system. It is not necessary to recompute and change all of the existing semaphore queue priorities throughout the system. Rather, it is only necessary to search over the possible semaphore queue priorities for the new task being added and to assign its priorities such that none of the existing tasks become unschedulable. A similar run-time schedulability determination is required with FIFO or RMSS when unexpected tasks are added, but these methods only try one fixed global semaphore priority for the task, so they may not be able to schedule the new task whereas the more flexible approach can.

An important advantage of rate monotonic scheduling over alternative scheduling approaches such as earliest deadline first is that rate monotonic scheduling behaves more predictably under overload conditions. This advantage is derived from the static nature of the rate monotonic priority assignment. If priorities change dynamically as deadlines approach, it is difficult to ensure that semantically important tasks will always be completed under overload conditions. Fortunately, both FIFO and BINP also use static priorities, so they share the predictability advantages of RMSS.

4.8 Summary

In this chapter, we have shown that the schedulability of real-time multiprocessor applications can be significantly improved if synchronization blocking delays are distributed according to task blocking tolerance rather than some fixed priority scheme. Often, higher-priority tasks that share global semaphores on multiprocessors should be given low global semaphore queue priorities. However, there is no fixed relationship between task execution priorities and semaphore queue priorities. We have analyzed the problem of selecting global semaphore queue priorities for real-time tasks on multiprocessors and have proven that this problem is NP-complete. Fortunately, it is essentially an NP-complete bin packing problem, for which heuristic algorithms perform quite well in practice. We presented such an algorithm and compared it with the RMSS method and FIFO scheduling on a large number of task sets.

Of the methods tested, BINP performed best by a wide margin. The next best method was FIFO, followed by RMSS. It is surprising that a simple FIFO performed better for real-time scheduling than RMSS, which is the best method for local semaphores. However, we have shown that remote blocking in multiprocessors is fundamentally different than local blocking in a uniprocessor. In multiprocessors, it is often better to assign more remote blocking to the more frequent, higher-priority tasks. Because FIFO distributes more of the blocking to high-priority tasks than does RMSS, it usually performs better.

The ultimate question of which method is best for real systems cannot be answered without reference to a particular implementation and a particular application. However, a real-time operating system could provide priority queues for global semaphores, default to FIFO priorities, and allow an application to choose different priorities during semaphore initialization. This would allow the application programmer to use whatever method is appropriate for that application.

CHAPTER 5
DEMONSTRATION

One of the key objectives of MDARTS is to support the development of real-time manufacturing control applications. To evaluate the suitability of the MDARTS design in this domain, we used MDARTS to implement a prototype motion controller for a six degree-of-freedom robotic manipulator. This chapter discusses the implementation and results of this demonstration.

5.1 Introduction

Manufacturing systems are often composed of customized hybrids of software and hardware from a variety of vendors. Customized systems are expensive to build and maintain, so efforts are underway to establish standard, open software architectures for advanced manufacturing. The proposed Next Generation Workstation/Machine Controller (NGC) for automated factories [2, 51] is representative of the trend toward open software architectures in manufacturing. The NGC is a software architecture specification for advanced cell-level machine tool controllers. The NGC architecture is designed for high-performance real-time computing platforms such as VME-based shared-memory multiprocessors. An NGC-compatible controller consists of multiple hardware and software components possibly supplied by different vendors.

Decomposition of software into separate tasks on multiple CPUs, as the NGC permits, introduces problems with respect to data sharing and communication. Shared data must be made accessible to, and used consistently by, all tasks that access them. It is also necessary to control concurrent access to prevent data inconsistencies. For real-time tasks, the data access should also provide strict real-time guarantees. Interestingly, the NGC architecture does not explicitly address the issue of real-time guarantees for shared data access (or for any of its specified operations, for that matter). A systems integration phase is proposed in which all such timing properties will be verified by some unspecified method. MDARTS, with its support for explicit timing guarantees, could facilitate the verification of timing properties in an NGC implementation.

MDARTS permits separate tasks in real-time systems to interact by performing operations on the common database. This is a very useful way to organize interactions, because it permits complex systems to be constructed around the database rather than in terms of direct inter-component interactions. The number of interactions can thus be conceptually reduced from $O(n^2)$ to $O(n)$. The database also provides consistent management of information that is of strategic value to multiple applications and subsystems in the overall system. These advantages of database systems are widely recognized, but without high performance and transaction-time guarantees such as provided by MDARTS, a database is of limited utility in hard real-time systems.

Rather than build an entire control system from scratch for our demonstration, we used MDARTS to provide a software interface to a commercial motion control board from Delta Tau Data Systems. The MDARTS object corresponding to the motion control board allows local or remote tasks to get and set fields in the object and thereby invoke the functionality of the motion control board. As the manipulator moves in real time, a higher-level control task on a Motorola 68030 host processor monitors the performance of the manipulator and supplies offset values to dynamically alter the path followed by the manipulator. To access the Delta Tau board, the controller on the 68030 uses a local MDARTS database object. MDARTS can easily meet the transaction-time requirements of this task (each of the transactions performed by the controller completes within 25 microseconds). The path followed by the machine can be programmed remotely using an X Window System interface on a Sun workstation. The remote interface uses the proxy object capability of MDARTS to query and modify the internal state of the controller across the ethernet. The results of motion experiments can also be immediately displayed graphically on the Sun workstation.

Although MDARTS was originally designed to provide real-time data management, this demonstration shows that the MDARTS framework for sharing objects and expressing timing constraints is useful in a much broader context. The MDARTS object that interacts with the Delta Tau device hardly fits the usual notion of a database object. However, the implementation of the object's get and set methods can be whatever the application programmer desires. Nothing in MDARTS limits these methods to data structure reads and writes. Furthermore, the timing characteristics of devices such as the Delta Tau controller can be reflected in the implementation of the MDARTS object.

5.2 Initial Experimental Setup

The machine we used in our demonstration was a six degree-of-freedom robotic manipulator made available to us by members of the Mechanical Engineering Department. Prior to our demonstration, this manipulator was used in adaptive assembly experiments for car body parts. The initial configuration of the machine and its associated computing hardware for these prior experiments are illustrated in Figure 5.1.

We will first describe the experimental setup used in prior experiments and then explain the changes we made to demonstrate the capabilities of MDARTS. Figure 5.1 depicts the hardware of the original system. It is a physical mechanism for geometric error compensation at the assembly stage of automotive applications. This mechanism consists of a multi-axis manipulating device (essentially a robotic table to which sheet metal parts can be affixed), a number of optical sensors which acquire the relative part positions and orientations, a XYCOM-682 (PC/AT286) computer which controls the overall system activity and provides the user's interface, and a multi-axis servo-motion controller which handles the execution of desired motions at the manipulator joint level. The servo motion controller board is a commercial product designed and manufactured by Delta Tau Systems. This Programmable Multi-Axis Controller (PMAC) has a digital signal processor (Motorola DSP5600), and it is capable of controlling up to 8 machine axes. The manipulator consists of a fixed base, a movable platform, and six independently positioned legs. Each leg is connected to the base by a 2-DOF joint and to the platform by a 3-DOF joint. The tops of adjacent legs are joined together at the platform connection point, forming a set of three

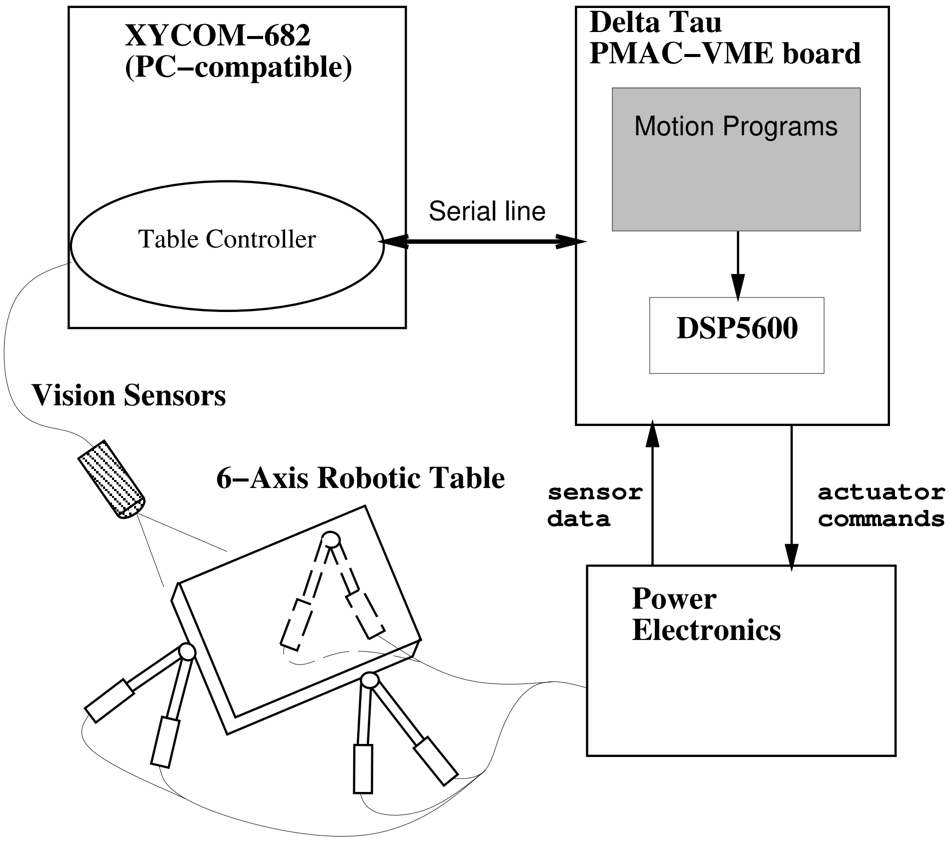


Figure 5.1: Initial experimental setup.

leg triangles.

As Figure 5.1 shows, the manipulator controller used two processors. The first processor, the Xycom-682, was connected with a serial line to the second processor, the Motorola digital signal processor on the PMAC motion control board. The PMAC contains its own local memory, signal processing circuits, built-in functions for PID servo feedback control and machine coordinate transformations, and interpreters for programmable logic controller (PLC) and motion control programs. Although the PMAC is a powerful device, capable of running stand-alone applications, the Xycom processor performed three important functions in the adaptive assembly experiments. First, the Xycom served as a host computer to download programs and provide a user interface for the application. Second, the vision sensors used for precise positioning of the parts were more easily interfaced with a PC, so the Xycom was used to read the vision sensors. Finally, the unusual kinematic features of the manipulator did not fit well with the coordinate system functions built into the PMAC. This is because unlike most robots, the axes corresponding to the actuators are not orthogonal, and their relative angles change as the platform moves. Therefore, the mechanical engineer who designed the original experiments implemented forward and inverse kinematic functions for the table in the C language and computed these values on the Xycom. In these experiments, the control program on the Xycom performed the following tasks:

1. initialize the PMAC and establish home positions for the motor encoders,

2. determine the approximate joint coordinates to move the platform (and the attached part) near the final destination,
3. download a motion program with these joint coordinates to the PMAC across the serial line,
4. send the PMAC a command to run the motion program,
5. use the vision sensors to determine what fine position adjustment is needed to precisely align the parts,
6. download a new motion program to achieve the final positioning of the platform and command the PMAC to execute it,
7. repeat the previous two steps until the desired alignment is achieved.

Although this system could precisely align the parts, its capabilities were limited by certain aspects of the hardware configuration. In particular, the serial line used to communicate between the Xycom and the PMAC limited the speed of information exchange between the two parts of the control system. This limitation made it infeasible to include the Xycom in tight feedback loops controlling the table motion. The system was forced to operate in a "think...move; think...move" mode both by the low communication bandwidth and by the slow speed of the vision sensors. Furthermore, the low communication bandwidth made it difficult to closely monitor the performance of the manipulator to detect and diagnose any errors that might occur during operation. When such errors occurred, typically one or more of the motors was automatically shut down by the PMAC. Recovery at this point required a complete restart of the system.

In addition to these hardware-related limitations, there were difficulties associated with the software interface to the PMAC. The PMAC is a very complex device, and configuring and programming it correctly is difficult. Delta Tau supplies PC-based software that provides a convenient interface to the PMAC, but this software does not support customized features such as the vision sensors or the specialized kinematic computations required for this mechanism. Therefore, the control software used in the assembly experiments bypassed the high-level Delta Tau interface and programmed the PMAC at a relatively low level. Furthermore, neither the Delta Tau interface nor the software developed for the adaptive assembly experiments provide any support for remote access across a network. In a factory application, it is very useful to be able to monitor and control assembly machines remotely, so this is a significant limitation.

5.3 MDARTS Demonstration Plan

When we decided to use the PMAC device, we were immediately faced with a serious problem. Namely, the PMAC is incapable of directly supporting MDARTS. MDARTS is a C++ library, and we had no way to link C++ objects into the execution environment of the PMAC. Therefore, we created an MDARTS object that interfaced with the PMAC device. This MDARTS PMAC object, which resided on the 68030 host processor, provided functions for initializing and shutting down the device, sending and receiving ASCII messages through the PMAC mailbox registers, and reading and writing regions in the dual-ported RAM. These functions were invoked through database writes and reads from corresponding "fields" in the object. The PMAC object acted as an agent to forward MDARTS requests on to the actual PMAC board. In effect, we extended the concept of an MDARTS object from being an interface to a data structure in memory to being an interface to an arbitrary system component. This was a significant conceptual leap, but it was very easy to implement. The original MDARTS architecture required no changes to accommodate this new object type. The built-in support for expressing and evaluating timing constraints and for providing remote access across the network were just as useful for a device interface as for a data structure interface. The PMAC object also explicitly reflected timing information relative to the PMAC device, which had no built-in functions for inquiring about timing behavior. By representing timing knowledge explicitly, it is possible to avoid propagating implicit assumptions about timing behavior into application code. When used in this way, MDARTS can be an extremely powerful tool for enhancing real-time applications.

Our objective in the demonstration was to highlight MDARTS capabilities while overcoming some of the limitations of the prior system. The demonstration was not intended to perform a complex task but rather to establish an open architecture framework upon which future control systems could be built. Therefore, we focused on supporting high-speed communication between a host processor and the PMAC device, providing network access to the controller, simplifying the application interface to the PMAC, and demonstrating modification of the manipulator motion from the host processor. By modifying the manipulator motion during execution of each move command, we demonstrated time-constrained database interaction. Furthermore, as the mechanism moved, the controller running on the host processor periodically examined the state of the machine and stored the motion history in the database. This also was a time-constrained operation. To show the networking capabilities of MDARTS, we implemented a simple X Window interface on a Sun workstation that sent commands via MDARTS to the demonstration platform. The application on the Sun performed read and write operations on an MDARTS proxy object, and these requests were automatically forwarded to the SDM, which performed the transactions with its local PMAC object. The networking support in MDARTS enabled us to quickly develop remote monitoring and command capabilities for the demonstration.

5.4 Modified Experimental Setup

Figure 5.2 shows the hardware configuration that we used in our MDARTS experiments. Notice that we replaced the Xycom PC host with a 68030 processor on a common VME bus with the PMAC. The PMAC-VME interfaces to the VMEbus as a slave device. Commands to the PMAC and responses from it are sent in ASCII through a set of 16 8-bit mailbox registers which are mapped into the VME address space. This communication interface was rather awkward and slow since message exchanges required interrupt handshaking across the VME bus to coordinate access to the mailbox registers. For example, even a trivial send/reply message sent across the mailbox registers required over 4 milliseconds. To bypass this relatively slow communication path, we upgraded the PMAC hardware to include 8K X 16 bits of dual-ported RAM mapped into the VME address space. This hardware configuration is more suitable for real-time control applications since the host processor can communicate with the PMAC across the VME bus, and the dual-ported RAM permits monitoring and modification of the manipulator state and PMAC control strategy at extremely high speeds (less than 30 microseconds, including the MDARTS procedure call overhead). On the 68030 host, we ran the VxWorks real-time operating system, which pro-

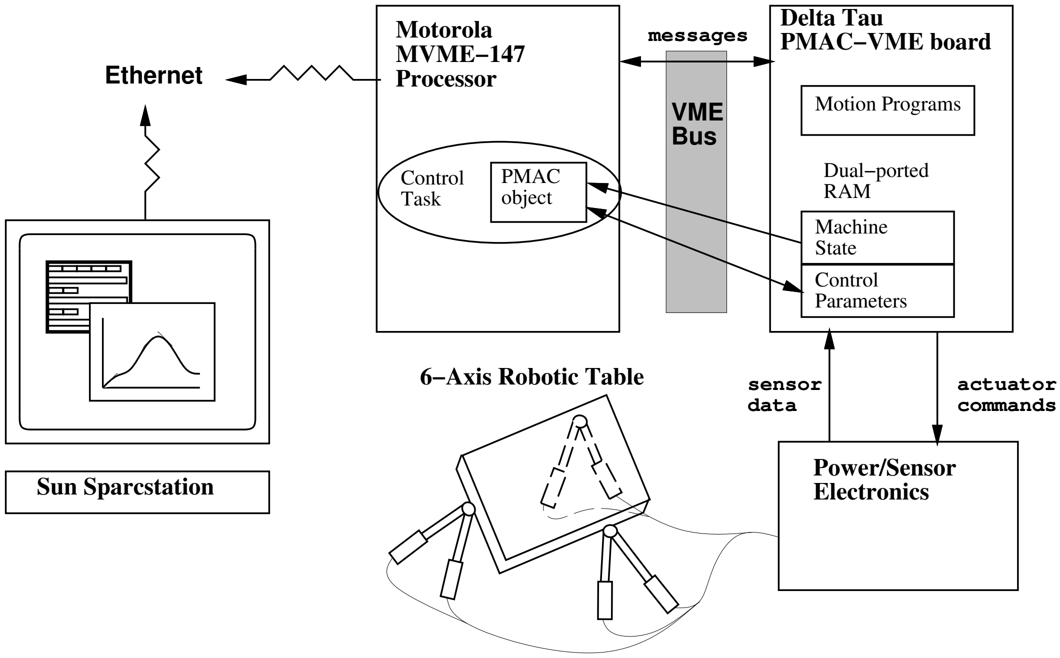


Figure 5.2: MDARTS demonstration hardware.

vides a multitasking real-time kernel with priority-based preemptive scheduling and support for TCP/IP networking with sockets or Sun RPC.

To demonstrate time-constrained use of MDARTS, we decided to modify the path of the manipulator while it was executing a sequence of move commands. In the adaptive assembly experiments, once the PMAC was programmed to perform a sequence of moves there was no way to dynamically modify its behavior. One could easily imagine a scenario where such dynamic motion control would be needed. For example, suppose the vision sensors detected an unexpected obstacle in the way of the programmed path. It would be useful to change the manipulator path at runtime to avoid the obstacle.

To perform dynamic motion compensation, we created two arrays of joint coordinates for one of the manipulator's axes. The first array was the original path for that axis, and the PMAC object used it to automatically generate and download a corresponding motion program to the PMAC device. The second array was a modified path for the axis such as might be planned by an obstacle avoidance algorithm. The modified path was not sent to the PMAC device but rather was used by the PMAC object while the machine was running to generate a series of offsets to modify the original machine path. The motion program executing on the PMAC processor added the current offset (which was kept in the dual-ported RAM) to the position specified in the original path. There was only one offset variable for the compensated axis, so the controller running on the 68030 host had to synchronize its execution with that of the PMAC motion program to determine which offset to supply at each point in time.

The PMAC motion program trajectories used in our experiments were linear blended moves. To achieve smooth blending of multiple move commands, the PMAC reads two moves ahead in the motion program. Reading one move ahead is necessary for smooth transitions in velocity, and reading a second move ahead is necessary for smooth transitions in acceleration. Because the PMAC processes the motion program two moves in advance,

```
static MdartsOb thePmac("Pmac","access<=30usec");

int modify_path()
{
    static int laststep = -1;

    int nextstep    = thePmac("program_step");
    if (laststep != nextstep) {
        laststep = nextstep;
        thePmac("path_offset") = thePmac("modified_path",nextstep) -
            thePmac("original_path",nextstep);
        }
    return 0;
}
```

Figure 5.3: Host controller code to generate offset.

our demonstration also had to supply the computed offset two moves in advance.

We tried to synchronize the executions of the PMAC and 68030 control tasks using a VME interrupt from the PMAC to the 68030, but this interrupt was already used to manage access to the PMAC mailbox registers. Using this interrupt for both purposes proved impractical, so we decided to poll the PMAC periodically from the 68030. We programmed a timer on the 68030 host to generate an interrupt every 20 milliseconds. In the controller interrupt service routine, we retrieved the manipulator's joint coordinates from the PMAC object and also retrieved the value of a counter that was incremented by the PMAC device each time it processed a motion command. From the value of the counter, the controller routine could determine which step in the motion program would be evaluated next and thus which offset to supply to modify the desired position for that step.

Figure 5.3 shows the C++ code used on the host to generate the offset for the PMAC motion program. The modify_path() function is called by the host controller's interrupt service routine. The 30-microsecond limit specified in the PMAC object declaration is somewhat artificial since our simple control program could actually tolerate a larger latency for each transaction. In the worst case, the controller on the host performed eleven transactions with the PMAC object. This controller executed once every 20 milliseconds, so on average each transaction had to complete in less than two milliseconds if the controller task were to complete before the next cycle. We specified tighter time constraints than this since we wanted to guarantee that the execution time of the interrupt service routine would be very short. The PMAC object could perform read or write transactions to the PMAC dual-ported RAM in less than 20 microseconds, so it could easily meet the 30-microsecond constraint.

Figure 5.4 depicts the X Window user interface we developed to run our experiments. From this interface, it is possible to initialize the PMAC, specify the original and modified arrays of joint coordinates for the motion program, run experiments, and display graphically the outcome (the actual vs. programmed motion of the machine). It is also possible to query the current axis positions of the manipulator at any time. Furthermore, the interface program can run on any Sun workstation on the network. All of the network communications between the interface and the controller on the 68030 host are handled by the MDARTS library.

To monitor the manipulator position during an ongoing movement, the host needed to retrieve the current manipulator position. One way to do this would be to query the PMAC command interpreter for the axis positions by sending ASCII messages across the mailbox registers. Unfortunately, the overhead to do this would be at least 4 milliseconds, which was too slow for our needs. Therefore, we programmed the PMAC to periodically place

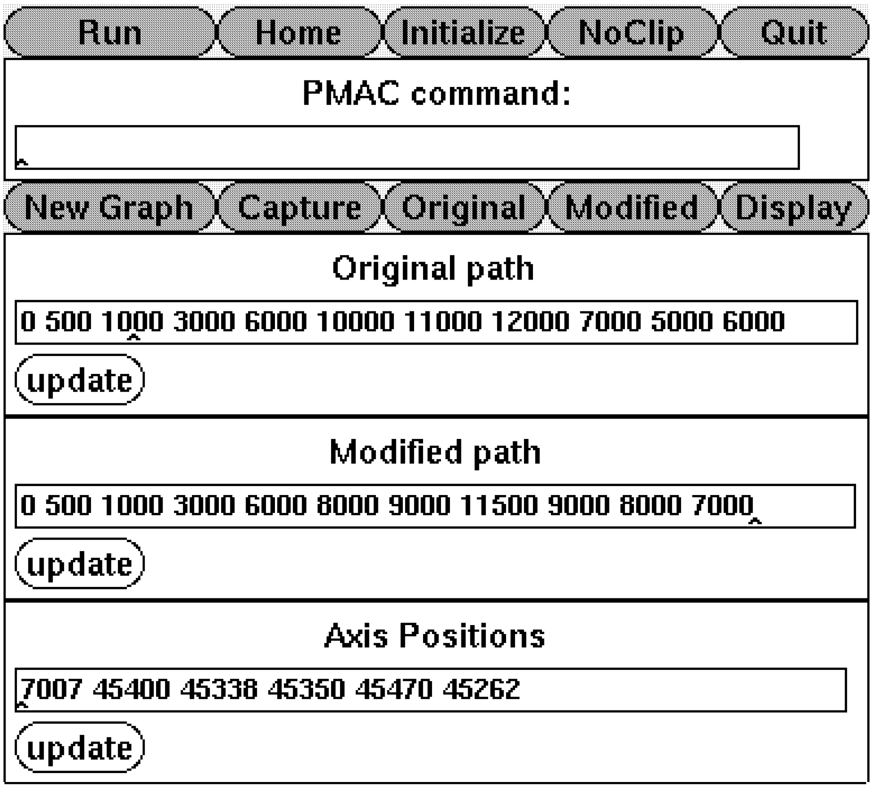


Figure 5.4: User interface for the demonstration.

the current axis positions of the manipulator into the dual-ported RAM. For this purpose, we used PLC program 0, which is a special-case PLC program that can be executed at the end of each servo interrupt cycle (every half millisecond) on the PMAC. To reduce the load on the PMAC, we configured PLC 0 to run once every 4.4 milliseconds since that was sufficient for our purposes. When queried for the axes positions, the PMAC object on the host retrieved them directly from the dual-ported RAM. The time required for the bus access to the RAM was at most a few microseconds. When the MDARTS procedure call overhead was added, the total transaction time was still less than 24 microseconds to read the axes positions. Figure 5.5 shows the result of executing and monitoring the test path illustrated in Figure 5.4. This graph overlays the original, modified, and measured path followed by axis 1 of the manipulator. Note that the measured path rounded off the sharp corner in the modified path to ensure a smooth transition between the blended moves. Otherwise, the measured path followed the modified path very closely. The paths do not start exactly at time 0 since it took about 200 milliseconds for the PMAC to start the motors. Therefore, we offset the original and modified paths in the graph to compensate for this delay.

As Figure 5.5 shows, our technique for modifying the manipulator motion in real time worked very well. It is also significant that we could perform these experiments with our X interface from any machine in the network.

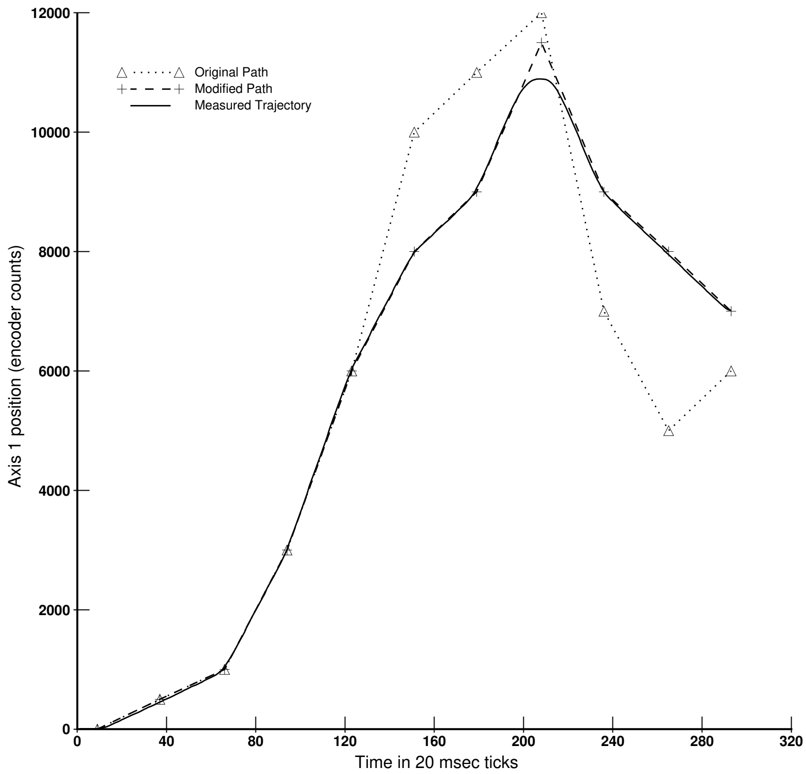


Figure 5.5: Demonstration.

5.5 Summary

This demonstration shows how MDARTS can be used to facilitate the implementation and integration of advanced manufacturing systems. In addition to providing local and remote access to shared data structures, MDARTS can be used to encapsulate the functionality of complex system components such as the PMAC motion controller board we used in our demonstration. The resulting interface reflected the timing characteristics of the component, and it supported remote access through the MDARTS Shared Data Manager server. This demonstration was very interesting from a systems integration perspective. We took an existing hardware platform that had significant limitations in functionality and radically improved its capabilities. We also designed a software interface to a physical device (the PMAC) and implemented that interface within the framework of our real-time database. By creating a PMAC object in our database, we provided both remote access and a convenient programming interface to a fairly complex hardware component. Finally, our X interface provided a convenient and powerful means of exercising the capabilities of the new system.

Not all of the MDARTS features were exercised in this demonstration. In particular, there was only one host CPU on the VME bus, so some of the multiprocessor capabilities of MDARTS were not shown. Furthermore, this demonstration did not attempt to perform very low-level feedback control such that maximum database performance was needed. In the future, we plan to apply MDARTS to a high-speed drill grinding application to fully demonstrate its capabilities.

CHAPTER 6

TIMING EXPERIMENTS

6.1 Introduction

Although the controller demonstration described in Chapter 5 illustrated many important features of MDARTS, it did not fully demonstrate the multiprocessor capabilities of MDARTS. The demonstration also did not rigorously test the timing guarantees provided by MDARTS objects. Even if we had implemented a more complex demonstration with multiple control and sensor monitoring tasks running on several CPUs, the timing and concurrency control features of MDARTS would not have been rigorously tested. This is because the critical section times for most MDARTS objects are so short that the rate of locking conflicts would be very low. A low rate of locking conflicts is good from the perspective of performance, but it does not test the behavior of the database under worst-case conditions, where contention is unusually high. Therefore, we conducted a series of experiments to evaluate the timing properties of various MDARTS objects in our prototype library. These experiments evaluated MDARTS performance under artificially heavy load conditions on a multiprocessor.

6.2 Experimental Platform

Figure 6.1 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, two 68020 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, and one of them served as a network gateway for the 68020 boards. The VME chips on the 68030 boards also provided support for inter-processor interrupts through VME write operations.

Note that this is a stock 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. Note that these boards were purchased in 1989. With modern processors, it would be possible to increase MDARTS transaction performance by

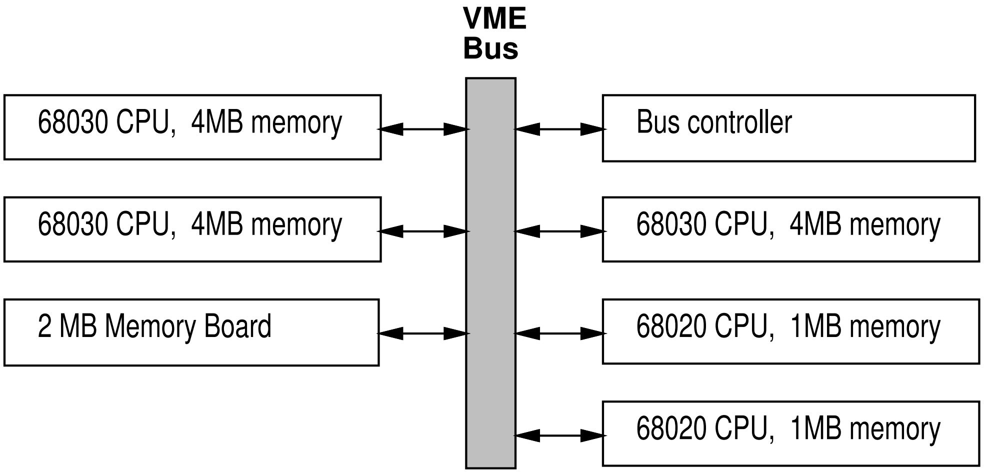


Figure 6.1: Platform used for MDARTS evaluation.

an order of magnitude or more. Clearly, the performance of transactions that use the VME bus are limited by the bus bandwidth, so one cannot guage system performance solely by the processor speeds. Nevertheless, the reader should keep in mind that the performance numbers reported here could be improved dramatically if newer processors and/or a faster bus were used.

The local memory on each processor board is visible to its own processor starting at address 0. This local memory is also mapped into a unique region of the VME bus address space. Therefore, remote processors can access the local memory of other processors by issuing VME read and write operations at the appropriate addresses. Requests for access to the local memory of a remote processor can encounter delays associated with contention for the remote processor's local bus. Unlike some shared-memory multiprocessors, this platform provides no support for local caching of global memory. Every read or write to the VME address space results in an actual bus request. For example, suppose processor P_1 has four megabytes of memory mapped in the VME address space at address 0x0D000000. In this case, an integer at address 0x400 on P_1 would be visible to processor P_2 at address 0x0D000400. If processor P_2 attemps to read an integer from this address, the hardware performs a rather complex sequence of operations. See Figure 6.2, which depicts the hardware components in more detail.

First, the read operation is generated by P_2's CPU on its local bus. P_2's VME chip detects that this is a VME request, so it issues a request for control of the VME bus. The VME bus controller determines when to grant P_2's request for the bus according to other activity on the bus and its bus scheduling protocol. Once P_2's VME chip gains control of the bus, it issues a read request at 0x0D000400. This address is recognized by P_1's VME chip as a request for access to P_1's memory. P_1's VME chip issues a request for control of P_1's local bus, which is granted by P_1's CPU. Once the local bus is controlled by P_1's VME chip, it issues a request to read an integer at local address 0x400. P_1's local memory responds and retrieves the data. P_1's VME chip then transfers the data to the VME bus and signal's P_2's VME chip that it is available. Finally, P_2's VME chip retrieves the data from the bus and supplies it to P_2's CPU across P_2's local bus.

Clearly, there is a fair amount of overhead in reading shared memory across the VME bus. However, the response time for a given bus request depends on the bus scheduling protocol and the current state of the bus and bus controller. If another CPU is using the

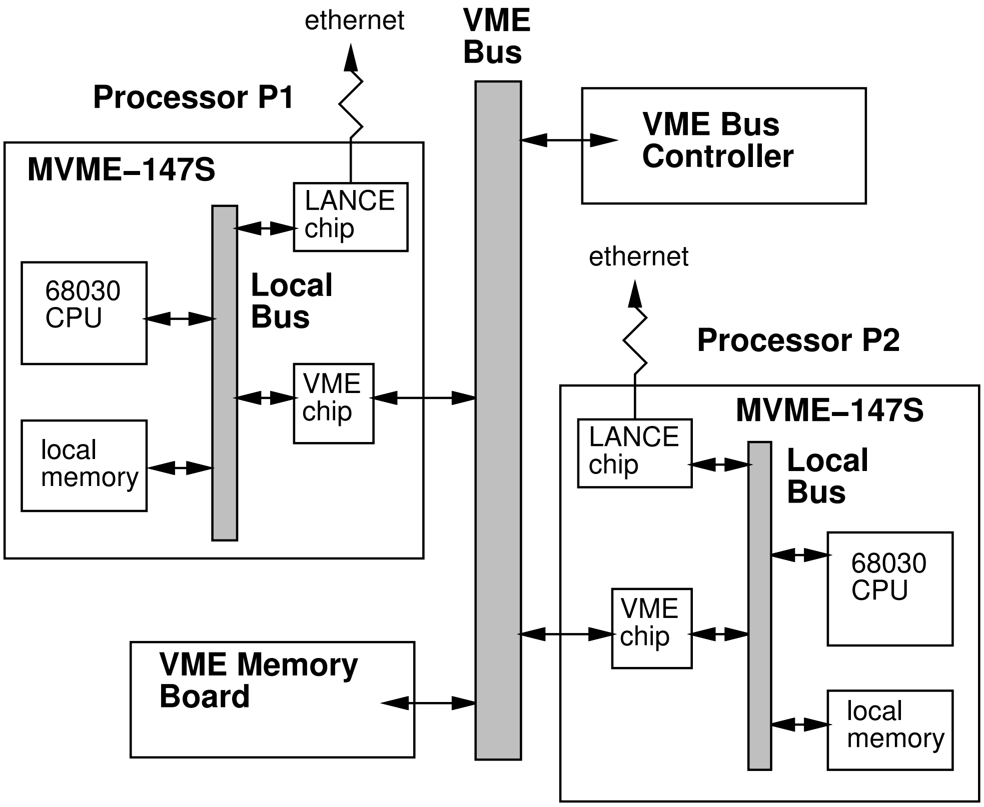


Figure 6.2: Detailed view of evaluation platform.

bus and other CPUs have requested the bus, a new request may have to wait several bus cycles for its turn. Another source of latency in reading another processor's memory across the VME bus is that the remote processor (i.e., P_1 in the above example) might be busy using its local bus when its VME chip requests access. These processor boards always give highest priority to the local CPU, so the remote request will be delayed until the CPU grants access to its VME chip.

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. In some cases, it might be preferable to locate the shared parts of MDARTS objects in processor memory rather than on a separate memory board. This is an implementation detail, but it will have an effect on transaction times. If the shared data structures of an object are located in one processor's local memory, transaction times for tasks on that processor will be faster than for tasks on other processors. MDARTS can determine during constraint checking whether the task creating an instance of the object has local access to the shared memory. If so, faster timing guarantees can be made for that task.

The response-time uncertainties in accessing shared memory are highly platform dependent. We did not select this platform to meet the needs of MDARTS. The platform happened to be available for our use, and we analyzed its properties and designed algorithms to match its characteristics. However, the platform-dependent parts of MDARTS are largely encapsulated in the implementation of a few functions in the Lock objects. Therefore, it would be fairly straightforward to design MDARTS implementations for different platforms.

6.3 Experiment Design

Given our hardware platform, we needed to design experiments that would exercise MDARTS and demonstrate its multiprocessor capabilities. 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. After trying a few different approaches, we decided to use interprocessor "mailbox" interrupts available on the 68030 boards to synchronize execution of competing transactions on multiple CPUs. The 68020 CPUs do not have hardware support for this interrupt, and they are also much slower than the 68030s. Furthermore, we did not want to complicate our experiments with the timing uncertainties associated with heterogeneous processors, so we performed our experiments exclusively on the three 68030 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.

From an application's perspective, MDARTS is a library of classes that can be used to construct shared objects that provide timing guarantees. Since each 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. Given this design, the implementation questions were how to construct the objects, synchronize their transactions, and collect the resulting timing information. We also wanted to have a flexible means of conducting multiple timing tests without compiling separate programs for each test.

6.4 Experiment Implementation

Figure 6.3 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 6.3). 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.

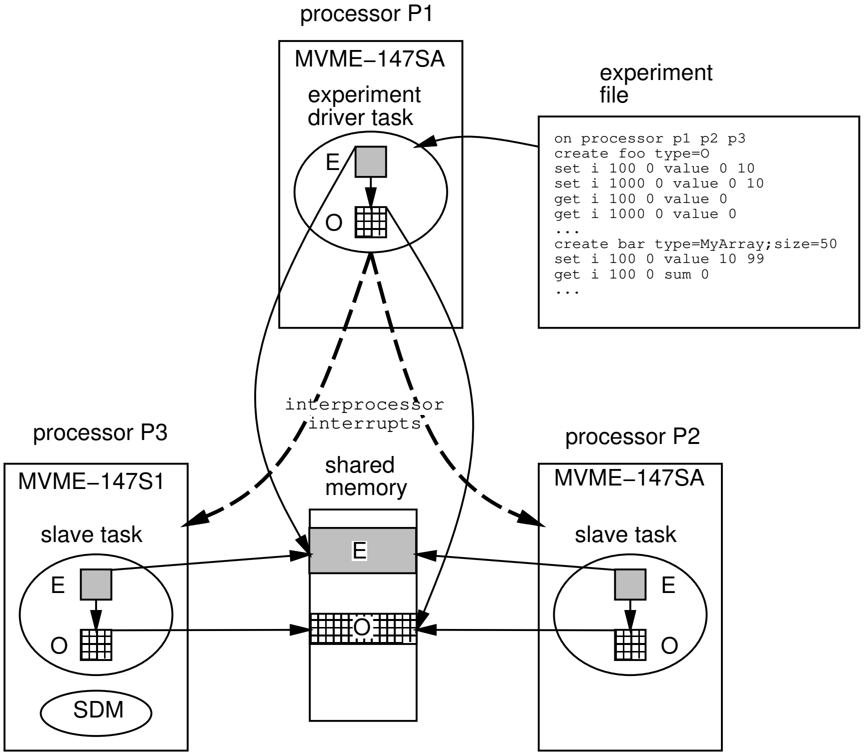


Figure 6.3: MDARTS experiments.

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. This signal was generated when the Experiment object's setValue method was called with a "start_experiment" tag parameter. When the signal was given, the three tasks performed their experiments in parallel and 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.

Figure 6.4 shows the code executed by the experiment driver task when it processed an input line such as: "get i 1000 0 value 0". This command caused parameters 1000, 0, "value", and 0 to be stored in the Experiment object's repeat_count, itag, tag, and index fields, respectively. Next, experiment type "i_get" was stored in the command field

```
void RunGetCommand(char *s)
{
   static RW_Mdarts ex_ob("experiment","type=Experiment");
   int count,itag,index;
   char tag[80];
   char type;
   if (sscanf(s,"get %c %d %d %s %d",&type,&count,&itag,tag,&index) != 5)
      ScanError(s);
   ex_ob("repeat_count") = count;
   ex_ob("itag") = itag;
   ex_ob("tag") = tag;
   ex_ob("index") = index;
   switch (type) {
      case 'i': ex_ob("command") = (int) Experiment::i_get;
      case 'd': ex_ob("command") = (int) Experiment::d_get;
      case 's': ex_ob("command") = (int) Experiment::s_get;
      }
   ex_ob("start_experiment") = 1;
}
```

Figure 6.4: Experiment driver code for getValue() transactions.

to indicate a getIValue() transaction. Once the parameters were stored, the experiment was started by storing a "start_experiment" command into the Experiment object. The repeat_count field specifies how many transactions should be performed for each experiment, so in this example, each CPU would perform 1000 getIValue() transactions. The transactions were performed using the MDARTS object O to which the Experiment object points. These transactions were performed in a tight loop, and the start time was synchronized by the "start_experiment" transaction. Therefore, this experiment executed a total of 3000 transactions (1000 on each of 3 CPUs). Each of the transactions was timed, and the worst case, best case, and total execution time were recorded by the Experiment object. Actually, the wall clock time rather than the execution time was measured in our experiments. If a task is preempted by another task in the middle of an experiment, the time read from the hardware timer will not reflect the true execution time of the transaction.

The code segment in the Experiment object that performs getIValue transactions is shown in Figure 6.5. This code segment is reached when the Experiment object's setIValue() method was called with a "do_command" value in its val parameter and an "i_get" value has been stored previously in the Experiment object's "command" field. 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 6.3) 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.

```
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++) {
     if (mask_it) intLock();              // possibly lock interrupts
        GET_COUNTER(time);
        int v = obj_ptr→getIValue(the_itag,the_tag,the_index); // experiment
        DELTA_TO_MICROSECONDS(time);
     if (mask_it) intUnlock();
     // 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 6.5: Experiment class method for getIValue() transactions.

6.5 Experimental Results

6.5.1 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 that integer with no locking. Although MdartsInt does not lock its data, locking a single integer is only needed for multiple-writer concurrency control. A single writer and any number of readers can share an MdartsInt object since 32-bit integer read and write transactions across the VME bus are atomic operations. MDARTS classes with more complex data, e.g., a character string or an array of integers, generally require more complex concurrency control. 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 6.1 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 6.1: Read and Write Wall Clock Times (in Microseconds) for MdartsInt Object.

Each row of Table 6.1 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. Only one bus operation was required for each “get” or “set” transaction. The “set” transactions tended to be slightly slower on average because of slightly more parameter passing overhead (the value to be stored is passed to the setValue method of MdartsInt).

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 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. 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 6.1 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. Since we were not running any other application tasks, the preempting tasks were operating system tasks, probably related to ethernet activity.

It is possible to prevent preemption during transactions by disabling processor interrupts when transactions are performed. Locking interrupts substantially reduces the variance in wall clock transaction execution times. With this technique we could eliminate the outliers and reduce the variance in execution times for the MdartsInt class to no more than one 6.25 microsecond clock tick. This technique is used when performing individual timing experiments for transaction calibration. However, locking interrupts can disrupt operating system activities. This is especially true if thousands of interrupt-locking transactions are performed in tight loops, as they are in our experiments. We quite frequently crashed the operating systems of our processors when running our experiments with interrupts locked. Furthermore, the apparent predictability gains of locking interrupts during transactions can be illusory. This is because as soon as the transaction completes and interrupts are enabled again, the task can be interrupted. From the perspective of a task performing a transaction, there is no difference between being interrupted during a transaction and being interrupted immediately after the transaction completes (provided the transaction itself is not affected by the interrupt or its handler routine). In short, we decided to allow interrupts during our experiments even though this decision made our worst-case transaction measurements appear longer than their actual execution time. Instead of locking interrupts, we counted the number of transactions that exceeded a certain threshold. Counting outliers does not completely characterize the execution time variance, but it does help distinguish rare events such as task preemption from true transaction execution time variance.

Another point to keep in mind when examining our execution-time measurements is that the granularity of our clock was about six microseconds. When averaged over 1,000 experiments, the effective accuracy of our measurements is better than 6 microseconds. However, the best and worst-case measurements refer to individual transactions for which the time granularity of our clock can be significant (particularly for very fast transactions).

In general, the best-case transaction times reported in our tables can understate the actual execution time by up to six microseconds. This is because a transaction that starts at time zero and executes for twelve microseconds will only see one 6.25 microsecond tick of the timer. It is also possible to overstate the actual time of a transaction by the amount of the clock granularity, but it is unlikely that an overstated time would be kept as a best-case value since this is the smallest measurement of the 1,000 transactions per CPU.

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 compromise 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 when fetching instructions, 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 (assuming the 68020 CPUs stay off the bus, which they should since they are idle in our experiments). The VME bus access delay is slightly unpredictable, but under almost all cases it will be no more than a microsecond. A VME system with more CPU cards will have larger worst-case latency, especially if bus scheduling is priority-based or if DMA operations across the bus are performed. For MDARTS classes to provide timing guarantees, it is necessary for the system designer to analyze the worst-case bus access latency and make that information available to MDARTS.

6.5.2 An MDARTS Array

Because the MdartsInt class does not use any locking for concurrency control, we performed another set of experiments on array classes that use 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 accesses. Furthermore, locking “get” and “set” transactions 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 was to experiment with different critical section lengths rather than trying to develop realistic transaction semantics.

We performed experiments on three different integer array classes. The only differences between these classes was the type of lock they used for concurrency control. Unless otherwise noted, the shared data regions of the array data structures and the locks were kept in a separate memory board on the VME bus rather than in the local memory of one of the processor boards.

```
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;
         tid = shared->lock.getLock();
            int the_size = shared->lsize;
            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 6.6: GetIValue() method for MdartsArray.

Figure 6.6 shows the getIValue() method for the MdartsArray class. Note how the itag is used to determine which field is requested of the object. If itag < 0, this means the application is querying the field number of the tag parameter. Converting a string tag parameter to a field number involves a series of string compare function calls, which are relatively expensive in execution time. A task with tight deadlines would not want to incur this overhead with every transaction. One alternative is to supply the itag field numbers corresponding to the desired field. However, a programmer may only know the name of the field it wants, not the field number. In this case, a task can first query the field number corresponding to a transaction it wants to perform. Subsequent transactions can be performed using this field number without incurring string comparison overhead. Our Experiment object uses this technique. Some tasks may be willing to tolerate the overhead of translating names to field numbers on each transaction. In that case, the field number in itag is set to zero and the field name is passed in tag. Given a valid field number, the getIValue() method performs the transaction and returns the result.

Figure 6.2 summarizes our experiments on the first of the three array classes. This class uses a spinlock queue that is designed to minimize bus traffic generated by tasks in the queue. For comparison purposes, we also include an experiment on a single CPU running out of its local memory rather than to the memory board across the VME bus. The local memory experiment is labeled 1^* in Figure 6.2. Note that data caching in the 68030 makes the local memory experiments even faster than one would predict from the nominal latency differences between local memory and VME bus memory. All of our spinlocks disable task switching when tasks enter critical sections. 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 anomaly was puzzling for a while, until I examined the physical hardware of the CPUs. The CPU board that we used for the third CPU was a 25 MHz 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. If CPU 3 happened to run the first transaction, or if it ran the last two, then at least one of its transactions would encounter no spin wait delays. In that case, its transaction time would be as fast as the single CPU case (faster, since the clock speed was faster). A similar observation applies to the best-case performance of the two CPU case.

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 6.2, one can see that on average the transactions required about 75 microseconds, but five transactions measured more than 100 microseconds. Since we are executing 1,000 transactions in these 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

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
1^*	get	50	55	218	3 > 100
	set	50	56	118	4 > 100
	size	12	18	181	1 > 100
	increment	62	67	137	0 > 170
	sum	62	64	131	0 > 150

Table 6.2: Wall Clock Times (in Microseconds) for MdartsArray with NQLock.

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 tasks, but we are not willing to be delayed 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 during 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. In fact, the 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.

Transaction Throughput

It is interesting to examine the raw transaction throughput of these 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, executes at 43,000 transactions per second. Table 6.3 summarizes the average throughputs of Table 6.2.

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
1*	18,000	18,000	55,000	15,000	15,600

Table 6.3: Throughput (in Transactions Per Second) for MdartsArray Transactions.

Table 6.3 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. Another factor made the speedup appear worse for "increment" and "sum." Namely, since these transactions take longer to perform in the first place, the number of preemptions by other tasks is higher for these transactions. This effect can be seen in the worst-case transaction times and the number of outliers for these transactions.

With only one CPU, none of the transactions experienced preemption (we know this since none of the worst-case transaction times were in the seven or eight hundred microsecond range typical of preemptions). However, with two and three CPUs, we observed some preemptions. These preemption-caused outliers increased the average transaction times. We only recorded the worst preemption delays, so we cannot determine precisely the magnitude of this effect. However, let us examine the "increment" transaction with three CPUs. We observed 26 preemptions on the three CPUs. At least one transaction was preempted for about 1,700 microseconds. We also know that often these preemptions take about 800 microseconds. If all but one of the preemptions took 800 microseconds, then the average transaction execution time was increased by about 7 microseconds (26 times 800 microseconds per 3,000 transactions). If all of the preemptions consumed 1,700 microseconds, this would increase average transaction time by 14 microseconds. Since preemptions caused the average transaction times in the three CPU case to be higher, the MdartsArray "increment" transaction will have a better speedup for multiple CPUs than is reflected in Table 6.3. Nevertheless, the mutual exclusion enforced by the locks limits the possible speedup. The larger the critical sections in relation to the overall transaction time the more speedup will be limited.

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.

By comparing the average-case performance of the different transactions on one CPU, it is possible to derive an estimate of the critical section times (direct measurements of critical section times are also possible). The "get" and "set" transactions have trivial critical sections, so their execution times can be taken as a baseline to estimate the procedure call and locking overhead of MdartsArray. If we subtract 75 microseconds from the average execution times of "increment" and "sum" transactions for the single CPU case, we derive 30 and 20 microseconds for their respective critical sections. However, the lock operations contribute about five microseconds to the critical sections, so we add five microseconds back to the execution time deltas to derive a more accurate estimate of the critical section times. Therefore, we conclude that the critical sections of "increment" and "sum" are 35 and 25 microseconds, respectively.

"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 6.3 supports this conclusion since the speedup is greater for "sum" than "increment" when a third CPU is used. However, these speedup limitations apply only to concurrent transactions on the same object. If an application issues concurrent transactions on different objects, no mutual exclusion delays are encountered, and the transactions can proceed in parallel. In this case, the bus may be the limiting resource in the system. However, the bus is fast enough to support very high transaction rates (roughly a million five-operation transactions per second).

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, system interrupts and task preemptions greatly increase the wall clock uncertainty of any given transaction. 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. If a remote lock-holding transaction is preempted, then this preemption time will directly impact the spin wait time of other tasks in the queue. This is a form of priority inversion, and our spinlocks disable task switching to limit it.

We have argued that spin wait delays attributable to remote 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. The most 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. Furthermore, local task preemption should not be charged to the preempted transaction. Our lock implementations prevent remote task preemption from blocking tasks on spinlocks, so this is not a problem.

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). The "size" transaction performs only one bus operation, so in the worst case it is delayed only one microsecond due to bus contention. The "get" and "set" transactions each perform only one bus operation in their critical sections, but their lock objects perform two or three bus operations in the critical sections, so in the worst case about five microseconds of latency will be added due to bus contention. The "increment" transaction performs two VME bus operations per array element (plus one operation to read the array size), and the "sum" transaction performs one per array element. In our experiments, the array size is 10, so the maximum cumulative bus access latency will be about 20 microseconds for "increment" and 10 microseconds for "sum" (assuming three CPUs and five VME 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,

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 6.4: Estimated Worst-case Performance for MdartsArray Transactions.

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 6.4 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 6.4 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. If Table 6.4 is compared with tables 6.2 and 6.3, 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. Actually, these worst-case estimates are probably slightly pessimistic. Remember that the average-case measurements are inflated somewhat by the inclusion of outliers caused by preemption, and the best-case measurements can understate execution times by up to six microseconds due to the granularity of our timer. Therefore, the actual 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 best and average-case times. This is a very desirable property for hard real-time systems.

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.

```
class NQLock {
    int theDataLock;
    int theCurrentNumber;
    int theNextNumber;
    char theWakeup[MAX_NUM_MACHINES];
    enum { stop, go };
    static char * gcsr[];      // global control/status registers
    static char * localcsr;    // local ptr to control/status register
    // ...
};
int NQLock::getLock() {
    volatile int x = 0;            // use x to add delays to tight loops
    int num;
    LOCK_TASKS;            // disable task switching
    *localcsr = stop;      // put this here to avoid race condition...
    while ( ! sysBusTas((char*) & theDataLock) ) {
        if (++x/3 > 1000) break;   // delay a few microseconds between checks
        }
            // counter lock is held until this indentation ends
            num = theNextNumber;
            theWakeup[num % MAX_NUM_MACHINES] = ThisCPUNumber;
            theNextNumber++;
            theDataLock = 0;   // release lock on counter
    BEGIN_CSECTION
    if (num != theCurrentNumber) {   // don't wait if we are next
        while (*localcsr != go) {    // spin on local flag in VME chip register
            ++x;                     // could also support a timeout...
            }
    }
    return num;
}
```

Figure 6.7: Implementation of NQLock spinlock queue.

6.5.3 Arrays with Alternative Spinlocks

The MdartsArray class discussed above uses a certain spinlock object to synchronize critical section access. The getLock() method of this lock is presented in Figure 6.7. This lock class uses the global control and status registers of the processor boards to avoid generating VME traffic during spin waiting. Each NQLock object keeps four variables in shared memory (the memory used by the lock is part of the shared memory region of MDARTS objects that use the lock). The first variable, theDataLock, is a lock used to control concurrent access to the NQLock object itself. The second variable, theCurrentNumber, is a transaction number for the current lock holder. The third, theNextNumber, is the next available transaction number. The fourth variable is an array of processor numbers corresponding to processors spinning in the queue. Note that task switching is locked at the beginning of the getLock() method. The BEGIN_CSECTION statement is a macro used to time critical sections during transaction benchmarking.

```
void NQLock::releaseLock(int my_number) {
    int next = my_number + 1;
    theCurrentNumber = next;
    if (next != theNextNumber) {   // bump the next one if someone is waiting
        *(gcsr[theWakeup[next % MAX_NUM_MACHINES]]) = go;
    }
    UNLOCK_TASKS;       // enable task switching again
    END_CSECTION
}
```

Figure 6.8: releaseLock() for NQLock spinlock queue.

When a processor spins in the queue, it spins on a flag in a local control/status register that can be set by other CPUs across the VME bus. This spinning does not generate any VME bus operations. When a task releases its lock, it checks the processor number array to determine which CPU should be granted the lock next and then sets the flag in that CPU's control/status register. The spinning CPU then exits the spin loop and uses the locked resource. Figure 6.8 shows the code executed when releasing the lock. In essence, this spinlock works like a "take a number" scheme in a candy store. A task that wishes to use the resource takes a transaction number and then waits until that number comes up. If it must delay, it does not keep checking the current number (which would generate VME traffic). Instead, it is notified by the task immediately ahead of it when its transaction number comes up. The design of these spinlocks is a bit tricky, but several good examples can be found in the literature (for example, [3, 19, 30]). This algorithm assumes that no more than MAX_NUM_MACHINES tasks will attempt to enter the lock. Since we disable task preemptions, this assumption is valid.

Since the performance of MDARTS transactions is so dependent on the concurrency control implementation, we experimented with two other spinlock implementations. The first is similar to NQLock except that it spins on the counter value rather than spinning on its local control/status register. This difference means greater VME traffic during lock contention but less overhead in the locking operations. The code for this lock is presented in Figure 6.9. Note that QLock's implementation is simpler and more portable than that of NQLock. However, the bus traffic generated during spin waiting could be significant in systems with large numbers of processors. Some shared-memory multiprocessors have hardware support for cache coherency. On machines like this, the spin waits would spin on

```
int QLock::getLock() {
    volatile int j,x = 0;
    LOCK_TASKS
    while (! sysBusTas((char*) &theDataLock) ) {
        j = ++x/3;          // delay a bit
    }
       int num = theNextNumber++;
       theDataLock = 0;   // release lock on counter
    while (num != theCurrentNumber) {    // spin until number comes up
       j = ++x/3;           // delay a bit
    }
    return num;
}
void QLock::releaseLock(int my_number) {
    theCurrentNumber = my_number + 1;
    UNLOCK_TASKS;
}
```

Figure 6.9: Code for simple spinlock queue.

the local cache of the counter until the task releasing the counter increments it. Therefore, this spinlock would not generate excessive bus traffic on some systems.

CPUs	operation	best	average	worst	outliers
1	get	62	67	187	17 > 100
	set	56	66	256	18 > 100
	increment	75	113	425	6 > 170
	sum	75	104	425	8 > 150
2	get	62	93	1175	9 > 150
	set	62	96	1775	11 > 150
	increment	75	151	937	29 > 200
	sum	75	129	1281	5 > 200
3	get	50	100	1250	40 > 150
	set	50	104	1787	41 > 150
	increment	68	166	2743	43 > 300
	sum	62	153	2712	29 > 300

Table 6.5: Wall Clock Times (in Microseconds) for MdartsArray with QLock.

The results of our experiments with MdartsArray and QLock are reported in Table 6.5. Note that the best-case transaction times are faster than those of NQLock, but the average transaction times are generally worse. This difference is attributable to the reduced VME bus contention of NQLock. As the number of CPUs increases, this difference becomes even more significant.

```
class SpinLock {
protected:
    int theDataLock;
public:
    SpinLock() : theDataLock(0) { }
    int getLock()
    {
        volatile int x = 0;
        LOCK_TASKS;          // disable the system scheduler
        while (theDataLock || ! sysBusTas((char*) &theDataLock) ) {
            int j = ++x/3;   // delay slightly between test-and-sets
        }
        return 0;
    }
    void releaseLock(int ignored) {
       theDataLock = 0;
       UNLOCK_TASKS;
    }
};
```

Figure 6.10: Code for an ordinary spinlock.

For comparison purposes, we also experimented with an ordinary spinlock that does not queue transactions. Figure 6.10 illustrates the implementation of this simple class. A task using this spinlock spins waiting for the lock until it acquires it, and there is nothing that prevents tasks on other CPUs from acquiring the lock multiple times while a previous task waits (i.e., this lock is not fair). As the task is spinning, it generates test-and-set operations across the VME bus. It is more expensive to execute test-and-set operations across the bus than the simple reads performed by QLock. Therefore, the spin loop first checks if the lock is free with a simple read before it tries to acquire the lock with a test-and-set.

The results with SpinLock are reported in Table 6.6. The simple spinlock performed best when there was no contention for the lock (the single CPU case). However, with multiple CPUs, NQLock performed better. The longer the critical sections and the more CPUs sharing the lock, the better NQLock performs compared with SpinLock. NQLock and QLock also have the advantage of being fair locks. It is impossible to determine analytically or empirically the worst-case transaction times of transactions that use SpinLock. This is because competing transactions can theoretically prevent a transaction from ever acquiring the lock. For this reason, NQLock is superior for hard real-time applications even in cases where the SpinLock would perform better on average.

Figures 6.11 to 6.13 summarize the measured throughputs corresponding to the three spinlocks. The throughputs of the different locks converge in Figure 6.12 since the length of the critical section of "increment" begins limiting the speedup possible through parallelism.

CPUs	operation	best	average	worst	outliers
1	get	50	66	281	9 > 100
	set	50	65	156	10 > 100
	increment	62	94	318	4 > 170
	sum	62	86	193	2 > 150
2	get	50	71	1800	9 > 150
	set	50	69	837	8 > 150
	increment	62	120	1231	10 > 200
	sum	62	114	1831	15 > 200
3	get	37	84	1325	22 > 150
	set	37	80	1750	18 > 150
	increment	56	170	1868	48 > 300
	sum	50	143	1856	31 > 300

Table 6.6: Wall Clock Times (in Microseconds) for MdartsArray with Spinlock.

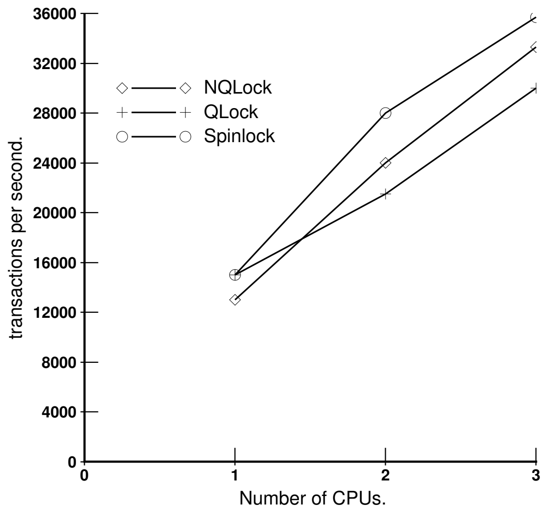


Figure 6.11: Measured throughput of the "get" transaction.

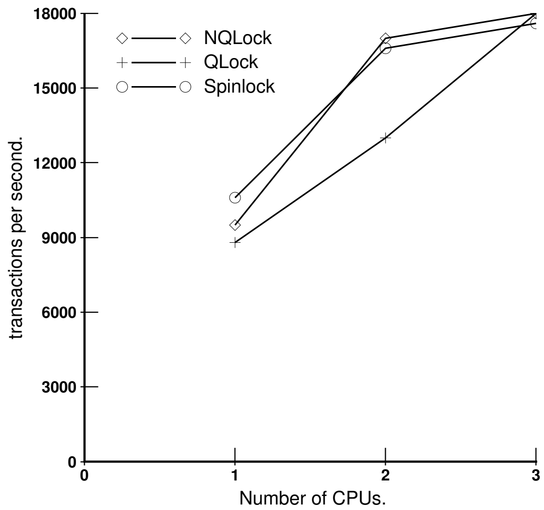


Figure 6.12: Measured throughput of the "increment" transaction.

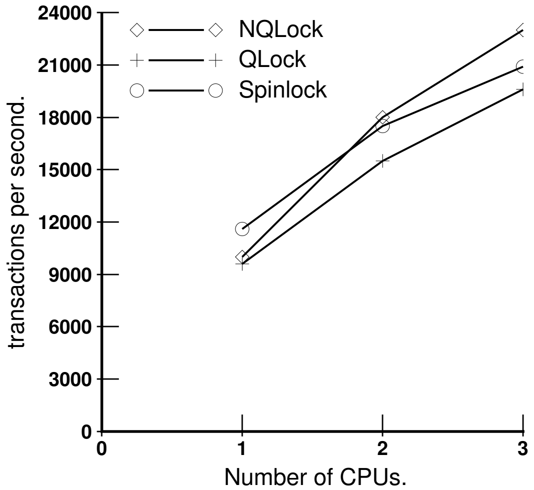


Figure 6.13: Measured throughput of the "sum" transaction.

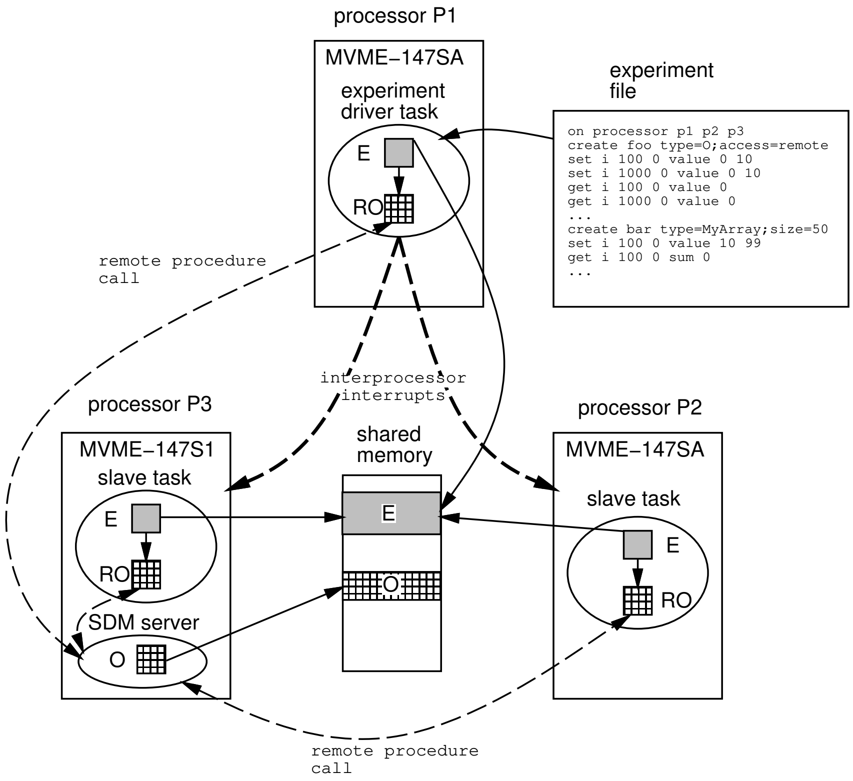


Figure 6.14: MDARTS experiments with remote objects.

6.5.4 Remote Access

We have presented the performance of direct shared-memory access by two MDARTS classes. 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 6.14 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. The objects we used in these remote access experiments were MdartsInt objects, which have trivial critical sections and no locking. 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, experiments with three CPUs necessarily ran one of the experiment tasks on the same CPU as the SDM. Table 6.7 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 6.7. 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.

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

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.

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 6.8. 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.

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

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 6.8. 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 6.7). 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.

6.6 Summary

Our experimental evaluation of MDARTS shows that with direct shared-memory access, MDARTS can support extremely high transaction throughput with worst-case transaction times that are very close to average-case. By avoiding context-switching overhead and communication delays implicit in inter-process communication between clients and servers, we can achieve about three orders of magnitude performance improvement for simple transactions. This can be seen when comparing memory-based MDARTS transactions with RPC-based transactions, especially if RPC clients and servers are on the same CPU. On a multiprocessor, MDARTS can provide nearly linear speedup, depending on the ratio of critical section times to overall transaction times.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

In this chapter, we review the contributions of this dissertation and discuss future directions for the research.

7.1 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 real-time transactions. Note that if high-performance remote procedure calls with worst-case latency bounds were available, the client-server architecture would become 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. Prior work on dynamic server selection through runtime service specification has been at the level of network services rather than local objects within processes [14]. 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 and analyze resource sharing delays by hand.

Finally, we have shown that in many cases multiprocessor schedulability can be improved by making higher-priority tasks wait longer on global semaphore queues than lower-priority tasks. In our experiments, we showed that a simple FIFO scheduling policy is usually better than using static task execution priorities for global semaphores. However, priority queues can provide better multiprocessor schedulability than FIFOs if the priorities are assigned according to the blocking tolerance of tasks rather than according to the task execution priorities.

7.2 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.

The current MDARTS prototype only provides spinlocks for concurrency control. This means that the transaction time guarantees of our implementation correspond to local task execution time rather than blocking time. For long critical sections, it is inefficient to use spinlocks for concurrency control, so semaphores should be used to block tasks waiting for shared resources. We have studied this problem in Chapter 4, but we have not yet incorporated this type of locking in MDARTS. To extend MDARTS in this area it would be necessary to develop new lock objects that use semaphores and to modify the transaction time specification methods to include both blocking time and local transaction execution time.

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.

Another area for future work would be to use networking protocols with end-to-end timing guarantees to provide real-time guarantees for remote transactions. This would require more than just predictable message latencies, though, because client transaction rates and server task priorities would affect transaction times.

MDARTS has an extremely simple application programming interface, so end users of the library can use it without knowing the details of how the classes or real-time guarantees are implemented. However, the developers of MDARTS classes must implement those classes correctly according to the protocols and structures of the MDARTS framework. It would be useful to develop higher-level tools to assist MDARTS library developers in creating classes that work correctly with MDARTS and accurately reflect their timing properties. For example, it would be possible to develop automatic test modules that exercise and attempt to verify the correctness of all the MDARTS methods defined by a class, including the transaction-time guarantees. A new class could be checked by the test module before it is included in the MDARTS library.

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. We have already taken the first steps in this direction in the MDARTS demonstration developed in collaboration with the University of Michigan Mechanical Engineering Department.

BIBLIOGRAPHY

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

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

[3] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, January 1990.

[4] 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.

[5] 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.

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

[7] B. R. Badrinath and K. Ramamritham, "Semantics-based concurrency control: Beyond commutativity," *ACM Trans. Database Systems*, vol. 17, no. 1, pp. 163–199, March 1992.

[8] T. Baker, "A stack-based resource allocation policy for real-time processes," in *Proc. Real-Time Systems Symposium*, pp. 191–200, December 1990.

[9] B. N. Bershad, *High Performance Cross-Address Space Communication*, PhD thesis, University of Washington, June 1990.

[10] J. Bloomer, *Power Programming with RPC*, O'Reilly & Associates, Inc., 1991.

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

[12] 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.

[13] 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.

[14] R. N. Chang and C. V. Ravishankar, "A service acquisition mechanism for the client/service model in cygnus," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 90–97, May 1991.

[15] 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.

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

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

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

[19] T. S. Craig, "Queueing spin lock algorithms to support timing predictability," in *Proc. Real-Time Systems Symposium*, pp. 148–157, December 1993.

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

[21] 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.

[22] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings, 1989.

[23] 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.

[24] M. R. Garey and D. S. Johnson, *Computer and intractability: A guide to the theory of NP–completeness*, Freeman, San Francisco, 1979.

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

[26] P. Gopinath and K. Schwan, ""CHAOS:Why One Cannot Have Only an Operating System for Real-Time Applications"," *SIGOPS*, vol. 23, no. 3, pp. 106–125, 1989.

[27] 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.

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

[29] M. H. Graham, "How to get serializability for real-time transactions without having to pay for it," in *Proc. Real-Time Systems Symposium*, pp. 56–65, December 1993.

[30] G. Graunke and S. Thakkar, "Synchronization algorthms for shared-memory multiprocessors," *Computer*, vol. 23, pp. 60–69, June 1990.

[31] C.-C. Han and K.-J. Lin, "Scheduling jobs with temporal consistency constraints," in *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 18–23, 1989.

[32] 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.

[33] R. Helm and Y. S. Maarek, "Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries," in *Proc. of OOPSLA*, pp. 47–61, October 1991.

[34] 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.

[35] 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.

[36] 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.

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

[38] D. D. Kandlur and K. G. Shin, "Design of a communication subsystem for HARTS," Technical Report CSE–TR–109–91, CSE Division, Department of EECS, The University of Michigan, October 1991.

[39] W. Kim and J. Srivastava, "Enhancing real-time dbms performance with multiversion data and priority based disk scheduling," in *Proc. Real-Time Systems Symposium*, pp. 222–231, December 1991.

[40] 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.

[41] 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.

[42] 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.

[43] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. Real-Time Systems Symposium*, pp. 261–270, December 1987.

[44] 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.

[45] 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.

[46] J. Liedtke, "Improving ipc by kernel design," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 175–188, December 1993.

[47] K.-J. Lin, S. Natarajan, and J. W.-S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," in *Proc. Real-Time Systems Symposium*, pp. 210–217, December 1987.

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

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

[50] 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.

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

[52] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[53] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," *Ph.D thesis*, 1983.

[54] 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.

[55] 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.

[56] 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.

[57] L. L. Peterson, N. C. Hutchinson, S. W. O'Malley, and H. C. Rao, "The *x*-Kernel: A platform for accessing internet resources," *IEEE Computer*, vol. 23, no. 5, pp. 23–33, May 1990.

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

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

[60] 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.

[61] K. Ramamritham and C. Pu, "A formal characterization of epsilon serializability," Technical Report CUCS-044-91, Department of Computer Science, Columbia University, 1991.

[62] K. Ramamritham, "Real-time databases," *Int'l Journal of Distributed and Parallel Databases*, pp. 199–226, April 1993.

[63] K. Ravindran and K. K. Ramakrishnan, "A model for naming for fine-grained service specification in distributed systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 98–105, May 1991.

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

[65] 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.

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

[67] 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.

[68] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[69] J. Shirley, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., 1992.

[70] 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.

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

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

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

[74] 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.

[75] K. P. Smith and J. Liu, "Monotonically improving approximate answers to relational algebra queries," in *Proceedings of IEEE Compsac, Orlando, Florida*, September 1989.

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

[77] 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.

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

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

[80] 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.

[81] X. Song and J. W. S. Liu, "Performance of multiversion concurrency control algorithms in maintaining temporal consistency," Technical report, University of Illinois, Urbana-Champaign, February 1990.

[82] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, October 1988.

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

[84] 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.

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

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

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

[88] 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.

[89] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *Proc. of OOPSLA*, pp. 227–242, October 1987.

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

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

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

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

[94] R. Wirfs-Brock and B. Wilkerson, "Object-oriented design: A responsibility-driven approach," in *Proc. of OOPSLA*, pp. 71–75, October 1989.

[95] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990.