Proceedings of the
American Control Conference
Baltimore, Maryland • June 1994

MDARTS: A Real-Time Database for the Control and Monitoring of Manufacturing Systems [1]

Victor B. Lortz Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
(313) 763-0391 {vbl,kgshin}@eecs.umich.edu

Abstract

In this paper, we describe an object-oriented memory-based real-time database system called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). MDARTS is specifically designed to support high-speed hard real-time applications such as next-generation manufacturing system controllers. MDARTS allows applications to specify their real-time requirements in application code, and during object initialization it attempts to guarantee that these requirements will be met. We have implemented MDARTS on Sun workstations and VME-based multiprocessors and have used our prototype to control an actual manufacturing machine. Our MDARTS prototype can guarantee transaction response times of about 100 microseconds for typical memory-based transactions on VME multiprocessors using 68030 processors.

1. Introduction

As real-time manufacturing systems become more complex, it is desirable to use a database system to manage data shared between different software entities (tasks, processes, modules). This shared database can store a wide range of information: e.g., part specifications, part programs, machine characteristics, control equation gains for machine axes, histories of performance data, and the current state of the machine(s). If all of this information is in a globally-accessible database, it can be used for both low-level servo control and for high-level supervisory control of manufacturing workcells. 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 central database.

The primary difficulty in using database technology to implement high-speed manufacturing system controllers is that these controllers are hard real-time systems, and conventional database systems do not provide the performance levels or response-time guarantees needed for this type of application. It is possible to use a conventional database system to maintain information such as production histories and part inventories, but it is not possible to use these databases directly within the low-level feedback loops of a machine controller, because their response times are simply too slow and unpredictable. Tasks in real-time controllers may need to execute several read and write transactions in less than a millisecond. Not even experimental main memory database systems reported in the literature can achieve these performance levels [1, 2].

Therefore, manufacturing system controllers have traditionally used ad hoc methods for data management. Control systems often keep data structures representing control parameters and the state of the system in memory as ordinary variables. When this information is local to the control tasks, it is inaccessible to other software mod-

[1] The work reported in this paper was supported in part by the National Science Foundation under Grants DDM-9313222 and IRI-9209031.

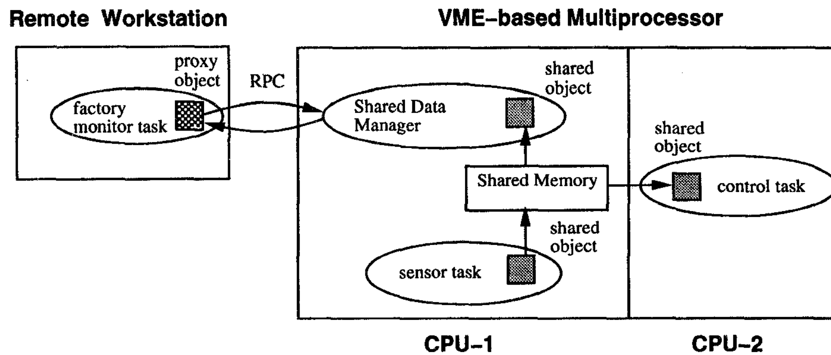

Figure 1: Access to shared memory data.

ules that might need it to perform execution monitoring or higher-level control. To permit more flexible data sharing, some systems make the memory addresses of certain data objects known to multiple tasks by using pointers or declaring global data structures at predefined shared memory addresses. Sharing data in this way is analogous to the "common block" feature of FORTRAN. As with common blocks, and for similar reasons, there is danger that a software component will inadvertently misuse or misinterpret the data and possibly corrupt the common data areas. Such errors are difficult to find and can have catastrophic consequences. In general, it is a bad idea to give independently-developed software modules raw pointers to common data areas.

In this paper, we describe a memory-based database system suitable for high-speed real-time manufacturing applications. Our database system is called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). A more detailed description of MDARTS may be found in [3]. Our MDARTS prototype is implemented in standard C++. It runs on Sun workstations or VME-based multiprocessors, and we have used it to control a multi-axis robotic mechanism in real-time. To our knowledge, MDARTS is the first real-time database system for multiprocessors that is capable of supporting hard real-time applications with transaction response times of less than a millisecond. MDARTS encapsulates access to shared memory using database objects, which are ordinary C++ objects that have been carefully implemented to support concurrent transactions. Since all manipulation of the data is performed by the object methods, application code never uses the raw memory addresses. The object methods ensure that the shared data is accessed consistently by all tasks.

The remainder of this paper is organized as follows. Section 2 presents an overview of the MDARTS architecture and application programming interface. Section 3 reports response times and throughput of shared-memory MDARTS transactions on a multiprocessor. Section 4 describes how MDARTS was used to implement a prototype manufacturing machine controller. Section 5 concludes the paper.

2. MDARTS Overview

MDARTS consists of one or more servers called Shared Data Managers (SDMs) and an object-oriented library of database classes. Real-time tasks needing to share data with other tasks declare objects belonging to the MDARTS database classes. These objects are automatically registered with an MDARTS SDM server that performs object lookup, allocates shared memory, and supports remote data access via remote procedure calls (RPC). Each MDARTS database class implements its own concurrency control protocol according to the semantics of its transactions (for a discussion of semantic and object-based concurrency control, see [4, 5]).

MDARTS fully exploits the hardware capabilities of shared-memory multiprocessors by supporting both remote network-based transactions and lo-

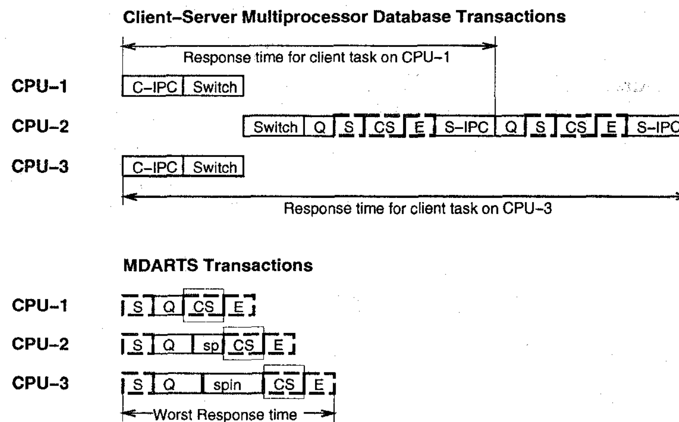

Figure 2: The advantage of avoiding a client-server architecture.

cal bus-based transactions. The locations and implementations of MDARTS objects are transparent to applications. Since the networking protocols we use (Sun RPC and TCP/IP) do not support response-time guarantees, only the bus-based transactions in our MDARTS prototype have guaranteed response times. A major difference between MDARTS and other real-time database systems (RTDBSs) is that local bus-based transactions on multiprocessors are executed by application tasks without communicating with a separate database server. This permits MDARTS to achieve much better performance and predictability than prior real-time database systems.

Figure 1 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 1). 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 transactions on behalf of the remote task. The object instances used by the SDM and the control and sensor tasks on the multiprocessor point to the same shared-memory region, so data consistency is guaranteed across the tasks. Mutual exclusion is provided through spinlock queues that use test-and-set instructions [6]. Spinlock queues cause tasks to busy wait, which is more efficient than blocking if critical sections are short. They can be either FIFO or priority queues. Note that once the shared MDARTS object is constructed, transactions performed by tasks with direct memory access require no inter-process communication. In this case, the MDARTS transactions are ordinary C++ function calls performed by the application tasks. Avoiding inter-process communication is extremely important. It is the primary reason MDARTS can achieve such high performance on multiprocessors.

2.1. Avoiding the Client-Server Architecture

Figure 2 contrasts the MDARTS approach with the usual client-server architecture for multiprocessor databases. 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

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

// Sensor task updates the data.  The update will take no more than 50usec.
//
position_sensors[5]  =  GetEndEffectorPosition();
```

```
//
// Corresponding declaration of MDARTS object in control task:
//
ReadOnlyMdartsArray<int> position_sensors("position_sensors",
   "read(element) <= 80usec; read(size) <= 40usec");

// Control task reads the data:
//
int end = position_sensors("size") – 1;

int end_effector_position = position_sensors[end];
```

Figure 3: MDARTS C++ application programming interface.

E. The relative lengths of these regions depend on the particular transaction. The client-server overheads are labeled as follows: **C-IPC** represents client-side inter-process communication, **Switch** represents context-switching (we assume that the server task on CPU-2 requires only one context switch to service both client requests), **S-IPC** represents server-side inter-process communication, and **Q** represents time required to enqueue client requests in the server.

The relative sizes of these overheads are not drawn to scale. The overheads depend on the characteristics of the target hardware and operating system. In most cases, the context-switching and inter-process communication overheads will be much larger compared to the transaction execution than Figure 2 implies. Although some operations are executed in parallel, the server remains a serial bottleneck in the system. In MDARTS, transactions proceed in parallel, executed directly by application tasks. Only the critical sections force serial execution and thus limit parallelism. Note that queueing of waiting tasks runs in parallel with the critical section of the lock holder. By avoiding client-server overheads, MDARTS can achieve extremely high performance on multiprocessors. Bus-based MDARTS transactions in our prototype implementation are two to three orders of magnitude faster than RPC-based MDARTS transactions.

2.2. The MDARTS Application Programming Interface

A unique feature of MDARTS is that it supports explicit declarations of real-time requirements and semantic constraints within application code. The MDARTS object constructors examine these declarations at runtime and create objects that are consistent with the application requirements (or signal a problem, if the requirements cannot be met). By registering application needs during object initialization, MDARTS is able to track resource allocation at runtime and guarantee response times before the transactions are actually performed. Prior RTDBS research has not considered the possibility of making response-time guarantees during initialization. From the perspective of a real-time task, each MDARTS transaction is an atomic operation with a bounded, worst-case execution time. Given worst-case execution times, it is possible to guarantee higher-level task deadlines through a straightforward application of real-time scheduling theory [7].

Figure 3 illustrates the MDARTS C++ application programming interface (API). The MDARTS classes in Figure 3 are MdartsArray<T> and ReadOnlyMdartsArray<T>, where <T> indicates a template class that is parameterized by an arbitrary class or structure T. In this case, T is simply an integer. Each object declaration specifies the

database name and a list of semantic and timing constraints. The MDARTS library verifies at run-time that the constraints will be met. The same MDARTS template classes that manages arrays of integers in Figure 3 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. Note that the MDARTS API is extremely simple, compared to embedded query languages typical of database interfaces to C or C++ programs.

3. Experimental Results

Table 1 shows the performance of our MDARTS prototype on a VME-based multiprocessor with three 20 MHz 68030 CPUs. The experiments reported in Table 1 used an MDARTS object that contained a 10-element integer array. Each CPU executed 1,000 transactions in each experiment, so the two and three CPU cases performed a total of 2,000 and 3,000 transactions, respectively. The transactions were performed in tight loops, and the start times of the experiments on the CPUs were synchronized. We used a hardware timer with a resolution of 6.25 microseconds to measure the execution times of individual transactions. The "get" and "set" transactions locked the object and returned or set the values of individual array elements. The "size" transaction returned the size of the array without locking the object. The "increment" transaction locked the object and added an integer to each of the ten array elements. The "sum" transaction locked the object and returned the sum of the ten elements. The relatively long critical section lengths of "increment" and "sum" limited the speedup (and hence the throughput) as more CPUs were added.

Occasional task preemptions and scheduler interrupts during transaction execution made it difficult to precisely determine the worst-case transaction execution times, so we report the average-case times. However, we deliberately created worst-case conditions, with the CPUs simultaneously performing a thousand transactions each on the same database object. Analysis of the transaction code and the VME bus access latencies shows that the worst-case transaction response times were very close to the times reported in Table 1.

CPUs	get	set	size	incr.	sum
average response times in microseconds					
1	76	75	23	105	97
2	83	82	25	115	111
3	90	86	25	167	129
throughput in transactions-per-second					
1	13,200	13,300	43,500	9,500	10,300
2	24,000	24,400	80,000	17,400	18,000
3	33,300	34,800	120,000	18,000	23,200

Table 1: Measured Performance of MdartsArray Transactions.

4. 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 motion controller for a six degree-of-freedom robotic manipulator.

Rather than build an entire control system from scratch, 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 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 response-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 X Window interface uses the proxy object capability of MDARTS to query or update the internal

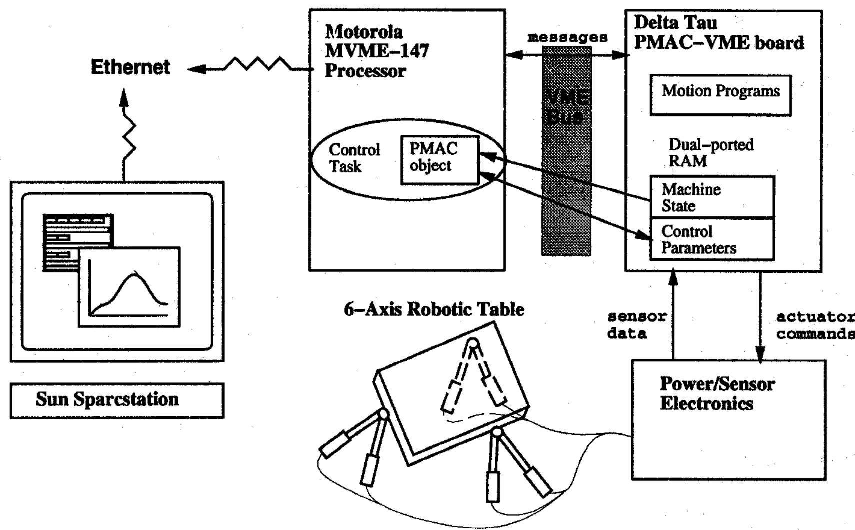

Figure 4: MDARTS Controller Demonstration Platform.

state of the controller across the ethernet. The results of motion experiments can also be immediately displayed graphically on the Sun workstation. Figure 4 illustrates the hardware platform used in this demonstration.

5. Conclusion

We have developed a new approach to real-time databases suitable for hard real-time control systems. The performance levels achieved by our prototype implementation are at least two orders of magnitude better than prior real-time database implementations. The primary reason for this performance advantage is that MDARTS avoids context switching and inter-process communication for bus-based transactions. Furthermore, MDARTS provides hard real-time guarantees, whereas prior real-time database systems do not guarantee response times and thus are suitable only for soft real-time applications. Our experiments show that MDARTS can provide excellent performance under extremely heavy load conditions. Furthermore, we have successfully used MDARTS to monitor and control a multi-axis robotic manipulator in real time.

References

[1] 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.

[2] 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.

[3] V. B. Lortz, *An Object-Oriented Real-Time Database System for Multiprocessors*, PhD thesis, University of Michigan, March 1994.

[4] 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.

[5] 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.

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

[7] 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.