A Multiprocessor System for Real-Time Robotic Applications

Mayez Al-Mouhamed

Department of Computer Engineering
King Fahd University of Petroleum and Minerals (KFUPM)
31261 Dhahran, Saudi Arabia

Abstract: The paper deals with the design and implementation of a multiprocessor, MRTA, having multiple module architecture. Each module consists of a tightly coupled set of processors connected to a shared resource by means of a multiport bus. A shared resource consists of memory, interrupt routing, and I/O for process control. The modules are connected through some of the member processors. With its modular architecture, MRTA is reconfigurable and expandable. Reconfiguration appears at two levels: the module interconnection topology and the type of interfacing of each shared resource. This architectures is suitable for multiple (parallel and sequential) and hierarchical formulations of real-time controllers, especially in the area of robotics. MRTA uses an efficient scheduling strategy to deal with large-scale problems such as robot dynamics. Perfor4mance evaluation was carried out to address the issues of scheduling, process monitoring, and synchronization. The variation in task execution time, which causes loss of synchronization, was studied together with its implication on the process finish times. It is shown, using and example that the resulting loss of synchronization has a reduced effect on overall performance when transfer time is moderate compared with execution time.

Multiprocessor system for realtime robotics applications

Hardware expandability and reconfigurability are key factors in the evolution of robot control systems. **Mayez Al-Mouhamed** presents a modular multiprocessor architecture, with each module based on a tightly-coupled 8086–8087 pair

The paper presents the design and implementation of a multiprocessor, MRTA, having a multiple module architecture. Each module consists of a tightly-coupled set of processors connected to a shared resource by means of a multiport bus. A shared resource consists of memory, interrupt routing, and I/O for process control. The modules are connected through some of the member processors. With its modular architecture, MRTA is reconfigurable and expandable. Reconfiguration appears at two levels: the module interconnection topology and the type and interfacing of each shared resource. This architecture is suitable for multiple (parallel and sequential) and hierarchical formulations of realtime controllers, especially in the area of robotics. MRTA was built to enable performance evaluation of realtime controllers. The system software was developed to allow inputting of the control problem, i.e. a set of tightly-coupled equations, and the generation of the parallel algorithm and codes for mapping within MRTA modules. The system uses an efficient scheduling strategy to deal with large-scale problems such as robot dynamics. To demonstrate the performance of the system, a typical problem of robot control has been investigated. Performance evaluation was carried out to address the issues of scheduling, process monitoring and synchronization. The variation in task execution time, which causes loss of synchronization, was studied together with its implication on the processor finish times. It is shown, using an example, that the resulting loss in synchronization has a reduced effect on overall performance when the transfer time is moderate compared with the execution time.

multiprocessors realtime systems robotics scheduling

The design of robot controllers includes both low-level control of the dynamics as well as higher-level intelligent decision-making, object understanding and three-dimensional trajectory generation. The organization of the robot controller as a hierarchy of processing modules (Figure 1) has been investigated at different levels[1–3].

- The robot task is described in terms of its final state (objective level).
- The robot task is a set of object transformations (object level).
- The robot task is a set of robot actions (effector level).
- Conversion between desired attributes and process attributes (physical level).
- Elementary requirements of the process (signal level).
- Devices to act on and to sense the environment (machine level).

Lower levels usually perform regular calculations with intensive interaction (short cycle) with the robot and its environment, while higher levels evaluate refined global decisions with relatively slower interaction (long cycle). At all levels, parallelism can be useful to speed up process control to different degrees.

Minicomputers can be dedicated to the control of robot systems, but are expensive. Multimicroprocessors have the advantage of reliability, i.e. failure in one processor can be handled by software reconfiguration and the system continues to perform, although there is some degradation in performance.

A flexible interconnection structure would be of great advantage in robotics where hardware continues to evolve. Hardware expandability and reconfigurability are the key factors in research and development of complete robot systems. Investigations have been made of various functional levels in robotics, such as control, motion coordination, trajectory generation and intelligent levels. Each level can be seen as a compact system consisting of a small multiprocessor with high data connectivity and means of communication with other functional levels. Highly sequential structures with the most efficient formulation of kinematics and dynamics of articulated systems have been recognized. Most of these functions are iterative in nature and require the development of special software to identify their possible parallelism[4].

Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
Paper received: 3 February 1989. Revised: 31 August 1989.

0141-9331/90/05276-15

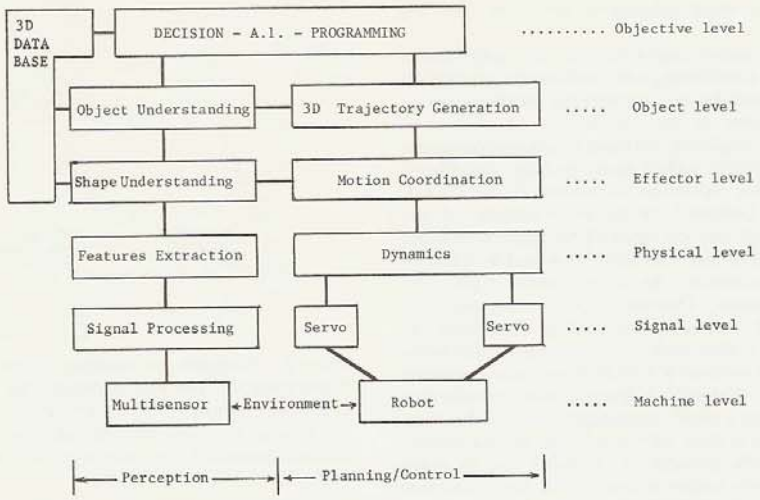


Figure 1. Control levels as a hierarchy of processing modules in a robot system

On the hardware side, portability, fault tolerance, power consumption and cost are other important aspects of robotics. Powerful VLSI chips, such as CPUs and numeric data processors, are required to simplify design and to improve processing capability and reliability. Special memory management software is required to reduce the overheads of executing the various functions concurrently. The extra cost of using fast VLSI devices is more than offset by the hardware simplification and increased system reliability.

Processor interconnections are of prime importance in designing multiprocessor systems. Loosely-coupled multiprocessors can be designed using network architectures. With multistage networks[5] and transputers[6], a processor interfaces with many other processors by means of datapaths. A datapath can either connect the I/O ports of many processors or be a common bus that routes many processors to a common memory. Generally, networks have great expansion capability and are consequently suitable for configurations involving a large number of processors.

Tightly-coupled multiprocessors are mainly shared memory multiprocessors (SMM), which are organized so that a processor may access the shared memory by means of a single[7] or multiple bus[8]. In the case of multiple-bus architectures, each processor-memory pair is normally linked by several redundant paths, implying greater fault tolerance. Thus an SMM generally has a limited neighbourhood but greater interprocessor data connectivity.

With regard to software development, most concurrent operating systems use the concept of variable sharing[9] to implement interprocessor communication and this can be directly applied to SMM systems. Variable sharing is more suitable for processors where high data connectivity is desired, i.e. processors at the same functional level of a robot system. The computation of dynamics on an SMM system involves hundreds or thousands of tasks having precedence relationships. The variable sharing concept offers an alternative to be investigated in synchronizing[2, 10] the computation of these tasks.

Multiprocessors with backplane buses[11] have a distributed common memory and a restricted interrupt-routing ability[7]. They are difficult to expand because of their restricted bus capacity and are less practical where one processor is to be interfaced to several buses.

Multiprocessors with multiport memories[12, 13] are easy to expand and to reconfigure. Their design greatly simplifies interfacing between the processors and common memories. On the other hand, they have no means of sharing the I/O and routing interrupts which are required for control problems. In realtime applications, the implementation of task concurrency is of prime importance. Task scheduling and synchronization require a detailed study of processor allocation and means of communication in a context where the tasks have variable execution times and strong precedence relationships. The use of commercial operating systems is not always possible if maximum benefit from the particular structure of the physical problem is desired. The initial decomposition of the problem into connective tasks depends on the scheduling method and the amount of transfer time. Both simulation and analytical results reveal that task size plays an important role in overall response time[14]. Communication between the various processors of a highly data-connective block can be designed using several methods, such as receiver polling or individual processor interrupts. The resulting overall performance is closely dependent on the nature of the tasks and the way they are synchronized. Therefore, the predefined communication tools that are commercially available do not necessarily lead to the best performance.

The implementation of parallel algorithms on multiprocessor systems requires scheduling software[15] to decompose the original control problem into a set of cooperating sequential processes that run concurrently on the available processors. The scheduling problem can be formulated as follows: a set of tasks with their constraints are to be scheduled on a given number of processors so that the overall execution time is minimized. This problem has been studied primarily as an optimization process for constructing optimal schedules[16]. The design of optimization algorithms has proven to be

very difficult as most scheduling problems are NP-complete[17].

The use of search algorithms as an optimization method[4] for the scheduling problem allows the generation of optimum schedules but requires the problem size to be small in terms of the number of tasks due to computational complexity. For most scheduling problems there exist heuristics, called approximation algorithms, which allow near optimum schedules to be found. Coffman and Graham[17] proposed methods of list scheduling which can be applied to graph-structured tasks, non-identical task execution time and an arbitrary number of processors. By using deterministic and stochastic problems, Thomas *et al.*[18] compared the performance of several list-scheduling algorithms. In particular, the critical-path or highest-level-first-with-estimated-times method (CP-HLFET) was near optimum in all the cases studied[18]. Generally, the optimization methods result in a small percentage saving in the total program execution time compared with approximation methods. The effectiveness of CP-HLFET and its polynomial complexity makes it one of the most effective methods to design parallel algorithms for large-scale problems.

This paper describes the design of a multiprocessor having a multiple module architecture for realtime applications, MRTA, together with hardware and software for the performance evaluation of robot systems. The modular architecture of MRTA and its basic components are described: processors, multiport buses and shared resources. The study discusses and compares the important steps in the hardware design and implementation, such as the generation of fast processor requests to the shared resource, decentralized arbiters and interrupt routing. These features have a significant impact on the multiport bus bandwidth. A system software module is outlined for scheduling large-scale problems in the context of precedence relationships and non-identical task execution times. The software allows high-level formulation of control problems, performs task generation, executes list scheduling and generates executable processor assignments. To demonstrate the system capability and resources in investigating the implementation of parallel algorithms, a typical robot control example is presented. Performance of MRTA software is first compared with other systems and then the effect is presented of variations in the task execution times on processor synchronization and overall system performance.

SYSTEM ARCHITECTURE

MRTA is a multiprocessor having a multiple module architecture (Figure 2). A module is a smaller multiprocessor with restricted neighbourhood interconnection. It consists of eight tightly coupled processors that are connected to a shared resource by means of a multiport bus. The modules are loosely connected through some of the member processors. MRTA consists of *P* processors linked by means of *B* multiport buses to *R* shared resources. Figure 2 shows one possible construction of MRTA.

A module has three basic components.

1 Asynchronous processors with private resource and bus extensions for interfacing with at most two multiport buses. To reduce bus contention, programs and private data are stored in the private memory of each processor. Synchronization of the process running on the processors can only be achieved with software.
2 Multiport buses. Each has an 8-to-1 interconnection network that allows a member processor of that multiport bus to access the shared resource with no bus contentions. The architecture of a multiport bus switch consists of a set of eight access ports (40 lines each) that is controlled by fast decentralized arbiters using a rotative priority scheme. The design of multiport buses improves the fault tolerance of the system with regard to processor failure.
3 Shared resource with a tailored configuration including memories and I/O. The memory is used for communication between the processors, i.e. software synchronization. The I/O devices allow the processors to share the control and the sensing of external processes and provide an interrupt-routing mechanism for exception handling and communication between the processors of different modules.

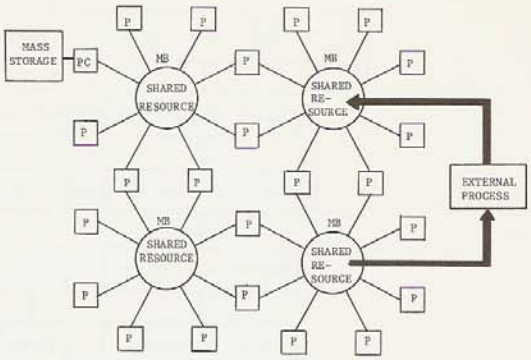


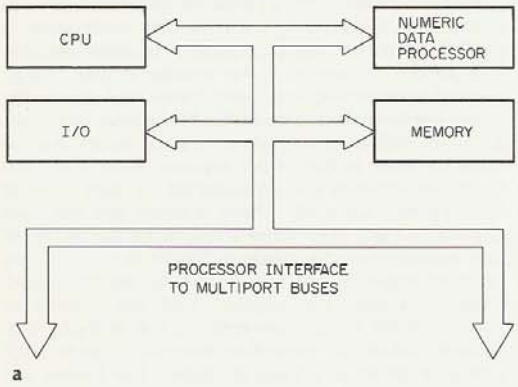
Figure 2. Examples of building blocks within MRTA. P, processor; PC, personal computer; SR, shared resource, including memory for local processor communication, I/O devices to allow processor sharing of the control of realtime processes and interrupts for intermodule communication; MB, multiport bus

At most, eight asynchronous processors can be tightly connected to a shared resource using one multiport bus switch. This construct defines one module. A processor can be a member of at most two modules, thus allowing modules to be loosely interconnected by means of processors. Two modules can be interconnected by means of two common processors. Depending on the amount of data transfer, a common processor may be either totally or partially dedicated to intermodule communication. Due to its modular architecture, MRTA is a reconfigurable and expandable system. This appears at two levels: the module interconnection topology and the architecture of the shared resource and its interface to the physical process. MRTA architecture is suitable for multiple, parallel and sequential, and hierarchical formulations of realtime controllers especially in the area of robotics. Direct mapping of a hierarchy of robotic functional modules, at lower and higher levels, to the multiple computing/controlling modules of MRTA can be achieved.

Processor resource

Each processor is made up of an Intel 8086-2 CPU (16-bit) and an 8087 numeric data processor (NDP) which run at 8 MHz (Figure 3a). This CPU has been successfully used for designing multiprocessor systems, such as DIRMU[12] and SHAMP[7]. One interesting feature of this CPU is that software development can be undertaken using IBM-compatible personal computers. Its cost performance ratio[7, 12, 11] justifies its use for research purposes.

The processor resources include 32 kbyte of read/write memory based on CMOS-6264 chips, 128 kbyte of EPROM memory based on 27 256 chips, a USART-8251, an interrupt controller PIC-8259 and a timer 8253. No wait-state option is required as all the components are accessible within 150 ns. Select logic is designed using three decoders 74S138, i.e. two decoders are used to generate memory chip-selects for even and odd addresses, respectively and one is used for I/O devices. Partial decoding is used also to optimize the number of on-board gate-level components. The processor architecture is organized around a private bus and two multiport bus interfaces. Optimum processor performance is achieved because no activity interference can occur between the private bus and the rest of the system. The multiport bus interface is used to connect that processor to two shared resources through multiport buses. This can be achieved by duplicating the processor's private bus interface over a number of 50 pin, 30 cm ribbon cables.



Processor request to shared resource

The CPU has two units: the execution unit (EU) and the bus interface unit (BIU). The EU executes instructions and the BIU fetches instructions, reads operands and writes results. The EU and BIU operate independently and are able, under most circumstances, to overlap extensively instruction fetch with execution. When no more fetches are required, the address available on the external bus is maintained for an extended time until completion of the current instruction by either the EU or the NDP. If simple decoding of the addresses is used to generate the processor request to the shared resource, then a processor may monopolize the multiport bus switch for a time which far exceeds that required to read or write data. To save this extra time, the chip select of a memory or I/O address within the shared resource is designed to become active, starting with a valid shared resource address, for four clock cycles. As all peripheral control signals complete their active phase within four cycles, this time (0.5 μs) is then sufficient to complete any read/write operation of a word (16 bit) from the shared resource.

Using this design, Figure 3b shows the timing of the chip select to shared resource (memory or I/O) when the CPU–NDP is executing the following program for loading three 32-bit real numbers from the shared resource to the stack of the NDP:

```
FWAIT FLD DW [ADDR0]
FWAIT FLD DW [ADDR1]
FWAIT FLD DW [ADDR2]
```

In Figure 3b CPU–ALE indicates that the CPU is setting a new address. SRA is the shared resource address, a low value of $\overline{\text{SRA}}$ indicates that a valid shared resource address is available on the processor bus. This address could be generated by either the CPU or NDP. RD is the CPU–NDP

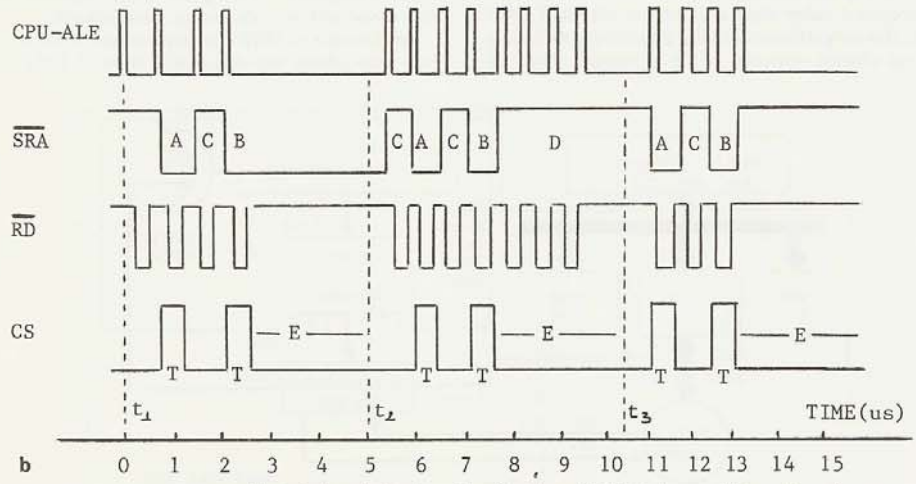


Figure 3. **a**, *processor resource and interfacing;* **b**, *timing of loading three 32-bit data words from shared memory (see text for description)*

read signal, CS is the chip select of a memory or I/O address within shared resource. Chip select is active from valid addresses ($\overline{SRA}$ is low) for four clock cycles. Chip select is used to generate the processor request for accessing shared resource through the multiport bus switch. The instants t1, t2 and t3 indicate the approximate starting times of the first, second and third instructions, respectively. The timing is labelled by using the following states.

- State A: the CPU is executing a dummy read of the low-order 16-bit word from the shared resource. These data are read by the NDP during its transfer on the data bus.
- State B: the NDP is executing a read of the high-order 16-bit word from the shared resource; the CPU is then waiting.
- State C: the CPU–BIU is executing a fetch from private memory while the NDP is storing the low-order 16-bit data onto the stack.
- State D: the CPU–BIU is fetching three words from private memory while the NDP is busy in state E.
- State E: the NDP is busy converting data (32-bit) to temporary-real format.

Following the first B state, the SRA is still active because no more fetch operations are required. According to the proposed design, chip select is only active (state B) for 0.5 μs, which is the minimum requirement with this CPU to accomplish the transfer. The sequence of states A–C–B indicates that overlapped fetch and execute operations are performed concurrently and, therefore, locked access to the shared resource may lead to extended access time because of interleaved fetches.

Execution of the shared resource statements causes the CPU–NDP to push the stack (NDP) and load from the shared resource the double words whose addresses are ADDR0, ADDR1 and ADDR2, respectively. Since each chip-select pulse has a duration of 0.5 μs, the effective bus time is the sum of these times, i.e. a duration of 3 μs if no bus contention is there. Note that the same load timing will be effective if the three operands were read from a private resource. With MRTA the request formulation to either a private or shared resource is identical and this process is transparent to the CPU-NDP; no extra overhead is required other than that of the standard CPU hardware. The only difference in loading/storing between private and shared resource is the potential effect of contentions in the latter case. The shared resource module behaves as a private resource with variable access time. As transfer of a 16-bit data word requires 0.5 μs between a processor and the SR, then the multiport bus allows data transfer with a bandwidth of 2 MW s^{-1}.

Multiport buses

A multiport bus provides a physical means of interfacing a number of processors to a shared resource with no contention. Transfers can be synchronous with one control, asynchronous with two controls, split-cycle synchronous, or split-cycle asynchronous. Split techniques give larger bus bandwidths and are complex to implement on today's microprocessors. Asynchronous transfers are more flexible but slower and should adapt for bus timing. In most cases, an arbitration logic is mandatory.

One exception to this rule is the 'memory offer addresses' (MOA) multiprocessor[19], which has no arbitration logic because the memory offers a sequence of addresses to the processors. Arbiters instantaneously solve the problem of selecting only one processor request by setting priorities. Depending on their architecture, arbiters can be either centralized or decentralized. A centralized arbiter treats all requests simultaneously, for example, a fixed-priority scheme that implements a parallel resolution using a priority encoder–decoder arrangement. However, if the priority should change depending on the requests, costly hardware is required to design the dynamic priority arbiter. Decentralized arbiters (DAs) are individually assigned to each access port to allow hardware polling of the requests. Serial or rotative resolution schemes can be designed by daisy-chaining the requests. Generally, these systems are not fault tolerant because they intercommunicate signals about bus availability by handshaking. Failure in one DA may cause shutdown of the whole system. On the other hand, delays in polling the requests limit the number of processors that can be connected to the multiport bus. Rotative priority is interesting because it gives each processor an equal chance to access. The lowest and highest priorities are assigned to the currently-accessing processor and next processor, respectively.

In the case of MRTA, an asynchronous scheme with a two-level arbiter was designed (Figure 4). Eight processor

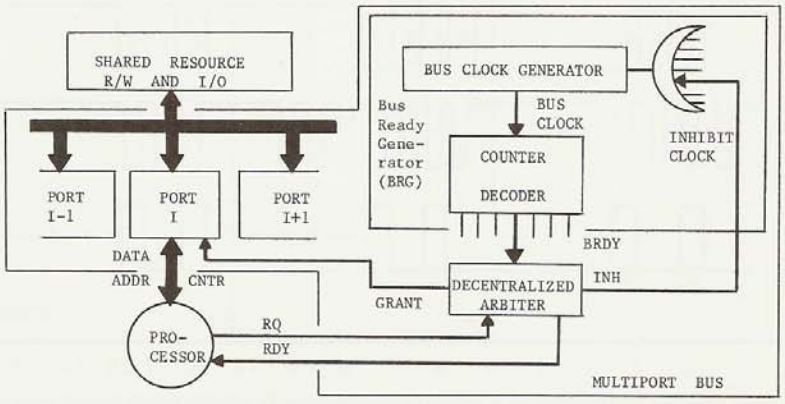


Figure 4. Block diagram of a multiport bus and its interfaces

access ports are available within each multiport bus. Each access port has a DA which synchronizes the request and sets the processor when it is requesting access into either state: accessing or waiting. The bus ready generator (BRG) uses a rotative priority scheme in polling (BRDY), the processor request RQ within each arbiter. The processor–port interface consists of 16-bit data, 16-bit addresses and eight controls.

In the first level the BRG uses a 32 MHz clock frequency with a counter–decoder arrangement to generate N bus ready outputs. For example, BRDY(I) indicates, at a given time, the bus availability from port I. To keep BRDY(I) active as long as request RQ(I) is holding, an inhibit line is used to inhibit the bus clock.

The second level consists of N DAs which are distributed among the access ports. A DA receives two asynchronous signals: the processor request RQ(I) and BRDY(I) from BRG. It generates three signals: bus grant G(I) to enable port I, processor ready RDY(I) and clock inhibit INH(I). To perform a transfer, a processor submits a request, RQ; the DA then responds by setting RDY according to bus availability. The signal RDY is connected to the ready input of the CPU, i.e. it allows the insertion of wait states into the processor timing whenever a request is holding and the bus is not available. Access grant G(I) is active whenever RQ(I) is holding and the bus is becoming available. The required DA equations are:

$$\mathrm{INH}(t+1) = \mathrm{RQ}(t) \,.\, (\mathrm{BRDY}(t) + \mathrm{INH}(t)) \quad (1)$$

$$\mathrm{G}(t) = \mathrm{RQ}(t) \,.\, \mathrm{BRDY}(t) \quad (2)$$

$$\mathrm{RDY}(t) = \mathrm{RQ}(t)' + \mathrm{BRDY}(t) \quad (3)$$

The timing of a typical transaction with a shared resource is shown in Figure 5. Another problem may arise when a processor monopolizes the multiport bus by making persistent requests, such as polling a parameter and consequently prevents access by other processors. This problem was studied by Nelson and Refai[20] who proposed a decentralized arbiter that allows only one bus transaction at a time with a shared resource. In this design, the method used to remove this undesirable feature is to anticipate the disabling of RQ when the CPU address is changing (Figure 5). To do this, the CPU–ALE pulse, which

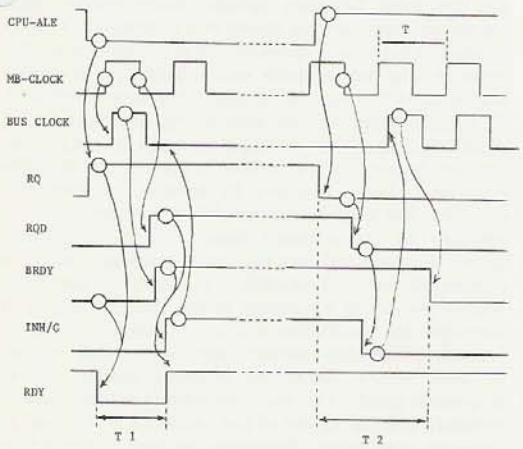


Figure 5. Decentralized arbiter timing (see text for description)

is 80 ns wide, is used to disable RQ, thus no processor can monopolize the multiport bus switch in case of persistent requests. The multiport bus clock is 32 MHz (T = 31.25 ns). The bus clock (BC)-to-bus ready (BRDY) delay is 16 ns maximum. RQD is the synchronized processor request. INH is used to inhibit the bus clock and enable the corresponding port. The processor ready (RDY) is an open-collector output which ensures compatibility when one processor is interfaced to more than one multiport bus. Wait states are only inserted in the processor timing when T1 > 240 ns.

The ALE duration is wide enough to allow the BRG to detect 'end of request' on the current RQ(I) and to start enabling the next BRDY(I + 1). In this case, at worst 62 ns are required to enable the hardware polling over the other bus ports. Thus a processor would only perform one bus transaction at a time when the LOCK logic is not used. The time from a valid bus request to the CPU's sampling of its ready input is $T1$ = 142 ns. However, the BRG cycle for polling the same port is $T2$ = 250 ns provided there is no other request. If only one processor is requesting the use of the bus, only one wait state (125 ns) will be added to the bus timing. This design preserves all the desired features pointed out by Nelson and Refai[20] and also avoids handshaking between the arbiters.

With this design, the CPU and NDP can access the shared resource and this requires at most eight cycles (1 μs) of multiport bus time to transfer a 32-bit data word (see above). The NDP internally requires additional time to convert the data to a temporary-real format, but this time does not increase the bus time of the MRTA design (Figure 3b), since both the CPU and the NDP require the same amount of time when transferring data with either private or shared memory in the case of no bus contentions.

On the other hand, a significant simplification of the interface is realized with this design as the processor reference with a shared resource is identical with that of a simple memory reference. This helps to minimize the processor–multiport interconnections, requiring only 40 interfacing lines for a shared resource, including 64 kbyte of memory and I/O.

Interrupt system

A multiport bus is capable of routing interrupts between any pair of member processors using a single 8-bit interrupt register (IR). All the member processors can write to and read from that register. Each output bit, I(I), of IR is directly connected, through port I, to the PIC input of processor I. To send an interrupt to processor J, processor I proceeds in accordance with the following interrupt program.

1 Request access to shared resource.
2 Read IR and test bit IR(J). If IR(J) is set, i.e. if an interrupt is pending, then exit from the shared resource and try again.
3 Access memory IP, write interrupt parameters, set bit IR(J), write the result to IR, and exit.

This method reduces processor overheads compared with other designs that either have limited interrupt routing[11] or require the processor to control the timing of the interrupt request[7]. The sending of interrupts requires several transactions with I/O and R/W memory within the

common resource. The arbiter described above treats processor requests on an individual basis, i.e. only one request can be honoured during each bus allocation. This presents a data coherency problem when dealing with interrupts, because the multiport bus should be locked throughout the interrupt sequence.

The 8086 has a LOCK signal which remains active throughout the execution of the instruction following a LOCK prefix. The LOCK signal can be used to coordinate accessing to the shared resource. However, if two or more instructions are each preceded by a LOCK prefix, there will still be an unlocked period between these instructions. To avoid this shortcoming, the circuit (Figure 6a) was designed to generate the processor request, RQ. The processor request RQ is a single line that combines a memory of I/O reference with the CPU-LOCK. Thus, a processor references a private or shared memory using the same instructions and timing in case of no bus contentions. Initially, the flip-flop output (Q) is reset following a hardware reset. When this flip-flop is in the reset state, the circuit output (RQ) is identical with chip select. In this case, the arbiter treats RQ on a one-request-per-bus-allocation basis. To execute the IR program, only the first and last references to the shared resource should be preceded by the LOCK prefix, thus setting the flip-flop and holding RQ until the occurrence of the last reference which should also be preceded by a LOCK prefix. As a result, the IR (Figure 6b) program obtains full allocation from the DA arbiter during its critical part.

1 The multiport bus is locked: a sequence of instructions, the first and last of which are preceded by the LOCK prefix. RQ is then active for the duration of the sequence.
2 Single reference to the shared resource with no lock.

This ensures proper routing of interrupts between any pair of processors sharing the same multiport bus. On the other hand, processors that are members of more than one multiport bus have special interrupt service routines to propagate the interrupt from one shared resource to another. Interrupts can thus be routed indirectly across the whole system.

Interprocessor communication

A command in a robot system involves several functions such as the calculation of dynamic torques, motion coordination, trajectory generation and vision processing. The processing of each of these functions can be seen as a set of closely connected terms with precedence relationships. Decomposition of the problem involves assigning each function to a highly connective multiprocessor, i.e. a set of processors that share the same multiport bus. One problem for the operating system is to find the best schedule including processor allocation, timing and synchronization. In most cases, the generated code, i.e. the processor assignment, is resident in the private memory of the processors.

A multiprocessor should provide means of intra- and inter-function communication. The common memory available in a shared resource has three sections for storing shared variables (SV), message routing (MS) and interrupt parameters (IP).

The way in which the communication paths are implemented is of prime importance. In the case of

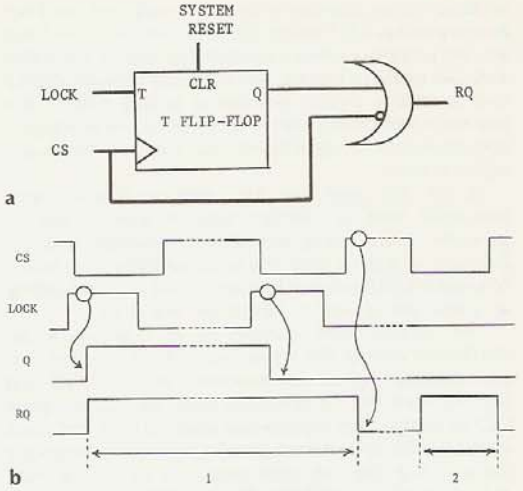


Figure 6. **a,** *processor request and LOCK logic;* **b,** *locked and unlocked request to shared resource*

intrafunction, the problem of communicating the value of a term to some other processors of the same multiport bus is simply the problem of one writer and several readers. The memory residing in the shared resource is adequate to support task communication for realtime problems requiring high data connectivity among the processors. Variable sharing, which naturally resides in the shared memory, is used for passing parameters among the processors. A task, the result of which is required to start other tasks, should have its value written to the shared memory and to ensure data consistency a version number is used. To avoid overwriting a value before it is acquired by the readers, the writer transfers the data and version number to a subsequent memory location every time it finds that not all the readers have read the data. Here, the tasks are known *a priori* and the number of readers is the same as the number of successors for that task.

Interfunction communication is less frequent but may involve more data. Message-oriented communicatic can be used between several multiprocessor sub structures. Messages are stored in a rotative FIFO buffer residing in the shared memory of each substructure. The message structure includes such information as sender and receiver IDs, function required and associated data. To simplify local message monitoring, two pointers are used to indicate the beginning of message storage and the first free location of the buffer. To transfer the message from one shared memory to another, at least one processor has to be a member of two multiport buses. In addition to computation tasks, this processor can administer message routing. Local message routing is performed using interrupts, i.e. once a processor completes storing a message in the shared memory, it interrupts the destination processors one after another. Member processors can then read the message with no software polling. Given the interrupt latency and its sequential nature, this message routing scheme would seriously limit the speed of the processor in the case of frequent operations. Therefore, its use is limited to operations involving massive data (downloading codes), starting of programs and emergency stop.

These communication tools represent the basis for synchronizing the processors in the context of tasks having precedence relations. This means that the computation of task T(*I*) assigned to processor K can start when its predecessor tasks have already been completed and their results acquired by processor K. The task data should also be moved from processor K to those processors assigned the successor tasks of T(*I*). In all cases, if at least one predecessor or successor is assigned a processor other than K, then data transfer is achieved through the shared memory. One function of the software is to add to task programs the communication routines and addresses that will all be determined by the end of the scheduling phase (see below).

SOFTWARE DEVELOPMENT

The design of robot controllers using a multimicroprocessor system allows an increase in both system throughput and in the sampling frequency of the process. Faster and more stable motion can thus be achieved by the mechanical system. In the field of articulated systems, behavioural model-based controllers can be described by means of iterative system equations defined over three-dimensional space. The Newton–Euler[4] and Lagrangian[21] formulations of dynamics become relatively efficient in terms of computation time when expressed according to their iterative form. These and other formulations are sequential at the equation level which poses a problem for their realtime computation. For example, the calculation of dynamics using the Newton–Euler formulation for systems with six degrees of freedom requires about 2 MFLOPS if a reasonable sampling frequency is desired[22].

The decomposition of those sequential equations leads to generation of a set of tasks with some precedence constraints. One important operation is to find a parallel algorithm for computing the tasks in minimum time. The process of scheduling the parallel computation of these equations depends on the architecture of the individual mechanical system. An investigation of the design of controllers for any mechanical system therefore requires the design of a scheduler to take advantage of the iterative structure, such as the generation of tasks which can be evaluated in parallel. The goal of such software is to identify the parallelism inherent in that mechanical structure and to generate the corresponding object code for multiprocessor implementation. The next section describes scheduling software which has been designed to generate efficient schedules for the computation of any iterative system equations defined over three-dimensional space within the area of robotics.

Task generation

The software admits a high-level input formulation of iterative equations including initialization, control and iterative sections. This organization allows the definition of the original problem in terms of starting conditions of the iterations, the operators and processors, and the iterative system equations, respectively. Generally, an equation consists of a number of fundamental vectors and matrix operations, the evaluation of which involves loading the operands, performing floating-point arithmetic and storing the results to memory. Shared memory is used to transfer data from the processor where the task is being evaluated to the shared memory, allowing other processors to read it and thus initiate the next tasks. Thus transfer of data to the shared memory only occurs for tasks with precedence relationships and different processor assignments. Concurrent requests to the shared memory are resolved by selecting one processor at a time and making the others wait. Frequent concurrent requests to the shared memory may lead to performance degradation of the overall execution time, i.e. a task synchronization problem.

Given the original set of equations, the task generation process consists of finding a criterium by which to decompose these equations into tasks that can be arbitrarily assigned to processors. There is no general criterium for partitioning a set of equations into tasks; this process depends to a large extent on the scheduling method. When search-based methods[4] are used for scheduling, the task generation process consists of a decomposition that satisfies minimum parallel-processing time within the limit of a reasonable scheduling time. In this case[4, 10, 22], the number of tasks and their sizes should be neither too small nor too large.

A different approach can be used when the scheduling method is applied to large-scale problems, i.e. constraint-free regarding the number of tasks and their sizes. In this case, task generation will be enabled to generate a larger number of tasks, provided that the time delay, caused by inter-task communication, is still reasonable. Assume a problem (see below) for which the software generates 140 vector operations and let each operation be defined as a separate task. In the worst case, the two predecessors and the successor tasks of each task T will all be assigned to other processors than T. In this case, T requires polling the version number of two operands, loading two vectors (predecessors) and storing one vector (successor). All vectors are defined in three-dimensional space. Then T requires the transfer of nine 32-bit operands and two 16-bit data words (version numbers) through the shared resource. Now, assume a minimum completion time of 5 ms for all the tasks (see below). Then a total of 2800 transfers of 16-bit data words will be needed. The required effective bandwidth will be 0.56 MW s^{-1}, in the worst case, which is only 28% of the available bandwidth (2 MW s^{-1}). The time spent computing a single vector operation using floating-point arithmetic is generally large compared with the transfer time of the operands through the shared memory. Table 1 shows the list of the fundamental vector operators in three-dimensional space. The columns of Table 1 list the operators, the computational complexity, operator time, loading and converting time, converting and storing time, overhead time, multiport bus time to transfer the operands and the result, the total time, and the ratio (*R*) of the multiport bus time over the total time, respectively. Note that the loading and converting time is the same when the data are loaded from either private or shared memory. The loading and storing times include the transfer time through the multiport bus switch. Here, we assume for each operator that the two operands are loaded from the shared resource and the result is also stored in the shared resource, i.e. the worst case. Table 1 shows that the vector–operator computation time is large compared with the transfer time through the shared resource as the ratio *R* ranges from 6.4% to 12%.

These factors encourage the decomposition of an

Table 1. Execution time of the vector-operators

Vector operators	Operations M: Multiply A: Addition	Operator time (μs)	Loading time (μs)	Storing time (μs)	Overhead time (μs)	Multiport-bus time (μs)	Total time (μs)	Multiport-bus time/total time
V × V	(6M, 3A)	104.625	22.5	28.125	2	10	157.25	0.064
M.V	(9M, 6A)	172.875	37.5	28.125	2	16	238.5	0.067
V.V	(3M, 2A)	57.625	22.5	28.125	2	10	110.25	0.09
V + V	(0M, 3A)	31.875	22.5	28.125	2	10	84.5	0.12

equation to fundamental vector-matrix operators and the creation of tasks with only one vector or matrix operation over two operands. Thus a generated task has two predecessors and an arbitrary number of successors.

The system proceeds by converting an equation to Polish notation and creating a number of new tasks to generate the original term. A new task is associated with every pair of terms combined by means of an operator. The operator is defined in accordance with the structure of the operands so that no operations containing zeros and ones will be generated in the object code of the latter stage. This optimization occurs whenever the structure of at least one operand is known in advance. Thus a generated task is defined completely by identifying its predecessors, operator, execution time and successors. The collection of these tasks and their precedence relationships allows a task graph to be defined.

Scheduling

The problem is to find a non-preemptive schedule for M identical processors which minimizes the processor finish times. Under these conditions, the scheduling problem is NP-hard in the strong sense[18]. No algorithm is available to generate an optimum solution for all cases.

Dynamic programming (DP) formulations of the task scheduling generate optimum schedules, but they require the problem size to be small with regard to the number of tasks, due to computational complexity. Luh and Lin[4] used a 'variable branch-and-bound' method for scheduling. They decomposed a system equation into a set of 88 tasks and divided that set into two subproblems because the number of tasks was too large to schedule. They applied the method to the computation of dynamics using Newton-Euler formulation and obtained a finish time of 9.67 ms for six processors; while 24.8 ms was required with one processor.

Kassahara and Narita[10] proposed a depth-first/implicit-heuristic-search (DF/IHS) algorithm for scheduling. To reduce the search process, DF/IHS allows the following setting for the upper bound of the acceptable relative error (E) of the solution (U) to the optimum solution (T_{opt}):

$$(U - T_{opt})/T_{opt} < E \tag{4}$$

At each branching, the heuristic used consists of selecting the ready tasks in the listing order of the critical path (CP) priority list[19]. Depending on the value of E, the method may generally produce approximate or optimal solutions. The behaviour of the algorithm becomes enumerative if the desired solution is to have significantly shorter execution time than that of the CP method, i.e. at least 5%. Using Luh's and Lin's equations, Kasahara and Narita obtained the optimum solution for seven processors while the processing power was 4.364.

With these methods the problem of task generation is controlled by the upper limit on the number of tasks with which the enumerative methods (EMs) can deal. Given the constraints on the number of tasks, the optimum solution obtained by using EM on a small number of large tasks does not necessarily correspond to the smallest execution time, particularly when the transfer time through the shared memory is small compared with the execution time. Performance could be improved if the original problem was decomposed into a larger number of tasks. The use of a larger number of tasks leads to an increased transfer time between the processors. Given a particular problem, the minimum computation time closely depends on the number of tasks, length of transfer time through the shared resource, amount of bus contention and the scheduling method. To investigate the computation time performance with a larger number of tasks than the EM can deal with, a near-optimum approximation algorithm (CP-HLFET) was used for scheduling tasks with precedence relationships and non-identical task times.

Critical path method

The critical path (CP-HLFET) method is an approximation algorithm consisting of the highest-level-first-with-estimated-times (HLFET) (see above). Coffman and Graham[17] evaluated the lower bound of performance for the CP method, i.e. finish time (T) in the worst cas compared with the optimum finish time (T_{opt}) and the number of processors (M):

$$T \leqslant (2 - 1/M)T_{opt} \tag{5}$$

This worst case bound does not show the true performance in an average case. As no formal proofs are available, Thomas *et al.*[18] compared several list scheduling methods using deterministic and stochastic problems and showed that the CP-HLFET is the most efficient approximation. The CP-HLFET was near optimum in all cases as it was not more than 4.4% away from the optimum solution. Application of CP-HLFET to a control problem involves the following steps.

1 Given the original system equations, the system creates the set of tasks by decomposing an equation into fundamental vector-matrix operations and generates tasks with precedence relationships,
2 Constructs the task graph (TG) and finds the number of successors S(T) for each task T.
3 Finds the task priorities or levels. The priority of a task is the longest path from any exit node to that task. The

search starts with the exit nodes or system outputs and progresses up to entry nodes. Whenever a task is allocated a priority its child nodes are expanded and the number of successors to these nodes is decreased by one. When exhausting S(T) for a task T, all occurrences of T in TG are then known and the selected path is the longest one. The priority is found for all the tasks if the original problem is well formulated, i.e. no loops. The system constructs the priority list in decreasing order.

4 Using the priority list, list scheduling is performed as follows: whenever a processor becomes free, the task with the highest priority among those not yet assigned is assigned to that processor.

PERFORMANCE EVALUATION

The objective of this section is to evaluate the effectiveness of the hardware and software systems in developing a parallel algorithm. To demonstrate this, a complex control problem was studied.

The controllers studied have large computational requirements, non-identical task execution times and strong precedence relationships. For scheduling, the task synchronization is based on the expected execution time. Indeed, the execution time of a task varies with the operands' values and the numeric data processor. In the case of the 8087 NDP, the variation[11] in the execution time for the arithmetic operators is of the order of 8%. The result is a loss in synchronization between the processors due to variation in the task execution time. In the following sections the performance of the scheduling method is compared with other methods, the performance of the generated schedules using a full implementation on MRTA is examined, and the impact of the loss of synchronization caused by the variations on the intermediate finish times is studied.

Comparison with previous scheduling methods

A typical control problem is used to investigate the performance of the proposed scheduling algorithm, CP-HLFET. The Newton-Euler formulation of dynamics for the Stanford Manipulator has been used by Luh and Lin[4] and Kassahara and Narita[10]. The objective of calculating the dynamics is to determine the driving force and torque at each joint of the manipulator. (The Newton-Euler equations can be found in Reference 4.) The scheduling software used allows the generation of the parallel algorithm, or task assignment for the processing units, for evaluating the Newton-Euler equations. To generate primary schedules, both groups of researchers[4, 10] assumed that the execution time for a multiplication is 50 μs, that for an addition is 40 μs, and the time for loading the data is assumed to be negligible. These times were input into the scheduling algorithm to determine the task size. Since each task is a combination of fundamental operations, its processing time may be determined as their sum. In the following, the same example and the same timings for the fundamental operations are used to generate the parallel algorithm based on the scheduling method described above. For this example, the scheduling software generates 139 tasks with precedence relationships. The number of processors M is an input to the system which partitions the tasks into a set of M programs called a schedule (S(M)). The scheduling operation is repeated for $M = 1, \ldots, 10$ processors.

Each schedule S(M) consists of a division of the problem under study into M programs. Given the set of tasks and their computational complexity, let FT(I), $I = 1, \ldots, M$, be the estimated execution time of each of these programs and let LFT(M) denote the largest among them. Thus LFT(M) provides the estimated longest finish time for the schedule S(M). The performance criteria is LFT(M). Among all the schedules (S(M)), let LFT_{min} be the shortest execution time.

To compare this approach with previous studies, Table 2 lists the schedules obtained by Luh and Lin[4], Kassahara and Narita[10], and this approach. LFT(M) and the processing power (PP(I) = LFT1/LFT(I)) are listed in this Table. This method gives LFT3 = 8.31 ms, while Luh and Lin obtained LFT6 = 9.67 ms. With 139 tasks, this method converges to the critical path solution for LFT6 = 4.85 ms, while with 88 tasks Kasahara and Narita obtained the optimum solution for LFT7 = 5.69 ms. PP6 is 5.113 in this method and 4.333 using Kasahara's and Narita's method.

Table 3 lists the schedules of the dynamics when six processors are used. This table shows how the 139 tasks are assigned to the processors and indicates that the longest finish time is obtained for processor 2. Table 4 lists the iterative tasks and their precedence relationships. These tasks result from decomposition of the original control problem into modular tasks to enable construction of the task graph. Table 5 lists the tasks of the critical path. The full implementation of the dynamics will be described in the next section.

Mapping of the schedules to MRTA

As shown in Figure 1, the design of the lowest control level of a robot arm requires the implementation of the system dynamics to generate the torques and N digital servos to regulate the motion of N robot-links.

The architecture shown in Figure 7 has three modules: M1, M2, and M3. The lowest level module is M1 which is interfaced to the robot through shared resource (SR) 1 and to module M2 by means of processors P3 and P4. Outputting of the commands for the control of the robot arm is achieved by using the I/O system within SR1. Six processors labelled S1, . . , S6 of M1 (S(I), $I = 1, \ldots, 6$) are used as servo-processors. Each S processor computes an independent proportional and derivative (PD) function[22] based on the state of the robot (I/O of SR1) and the

Table 2. Comparison with previous results

Processors	Luh and Lin[4] (LFT, PP)		Kasahara and Narita[10] (LFT, PP)		Proposed method (LFT, PP)	
1	24.80	1.000	24.83	1.000	24.80	1.000
2	—	—	12.42	1.999	12.41	1.998
3	—	—	8.43	2.945	8.31	2.984
4	—	—	6.59	3.768	6.35	3.905
5	—	—	5.86	4.237	5.28	4.696
6	9.67	2.565	5.73	4.333	4.85	5.113
7	—	—	5.69	4.364	4.85	5.113

Table 3. Scheduling the dynamics for the Stanford Manipulator

P	Terms	T(I,J,K)			
1	V0(4,4,0)	T1(4,32,0)	V0(5,36,0)	T1(5,64,0)	V0(6,68,0)
	T1(6,96,0)	V51(6,138,0)	V5(6,180,0)	T42(6,188,0)	T4(6,211,0)
	V14(6,253,0)	T71(6,261,0)	T7(6,265,0)	V13(5,273,0)	V10(5,293,0)
	T7(5,309,0)	V13(4,317,0)	V10(4,337,0)	T7(4,353,0)	V13(3,365,0)
	V10(3,389,0)	T71(1,409,8)	T8(2,433,20)		
2	V1(4,10,0)	V11(4,14,0)	V12(4,22,0)	T2(4,50,0)	V12(5,58,0)
	T2(5,86,0)	V12(6,94,0)	T2(6,122,0)	V6(6,164,0)	T41(6,172,0)
	V7(5,187,0)	T41(4,195,0)	T5(5,207,0)	T61(4,235,0)	V14(4,259,0)
	V6(2,283,0)	T4(3,306,0)	V7(3,321,0)	V14(3,331,0)	V7(2,346,0)
	T5(2,358,0)	V2(5,401,0)	V10(2,413,8)	T8(1,485,68)	
3	T2(3,10,0)	V1(2,15,0)	V11(2,19,0)	V51(3,38,0)	V51(4,71,0)
	V81(5,99,0)	V3(6,137,0)	V4(6,147,0)	T31(6,155,0)	T3(6,163,0)
	V8(5,192,0)	T6(5,204,0)	T42(4,212,0)	T4(4,235,0)	T6(4,247,0)
	V4(2,257,0)	T41(3,265,0)	V9(3,303,0)	T41(2,311,0)	T42(2,319,0)
	T4(2,342,0)	V14(2,366,0)	T71(2,378,0)	T8(5,382,0)	T8(6,386,0)
	T7(3,405,3)	T7(2,429,8)			
4	V2(6,28,0)	V3(2,66,0)	V1(6,76,0)	V11(6,80,0)	V6(4,104,0)
	V5(5,137,0)	T32(6,145,0)	T41(5,153,0)	T42(5,161,0)	T4(5,184,0)
	V14(5,208,0)	T71(5,220,0)	V7(4,235,0)	V5(3,254,0)	T5(4,266,0)
	T42(3,274,0)	T31(2,282,0)	T61(3,310,0)	T6(3,322,0)	T61(2,350,0)
	T6(2,362,0)	V9(1,386,0)	T6(1,405,7)	V13(1,441,24)	
5	T61(5,28,0)	V12(2,32,0)	V1(5,42,0)	V11(5,46,0)	T2(2,66,0)
	V51(5,99,0)	V81(6,150,0)	V8(6,210,0)	V8(4,239,0)	V81(3,267,0)
	T32(2,275,0)	T71(4,287,0)	T3(2,295,0)	V8(3,324,0)	T5(3,336,0)
	T71(3,348,0)	V6(1,353,0)	V5(1,363,0)	T42(1,368,0)	T4(1,373,0)
	V14(1,397,0)	T8(3,409,8)	V10(1,465,32)		
6	T1(2,10,0)	V51(2,43,0)	V81(4,71,0)	V5(4,104,0)	V6(5,128,0)
	V7(6,224,0)	T5(6,236,0)	V6(3,246,0)	V2(2,256,0)	V5(2,289,0)
	V81(2,317,0)	V8(2,346,0)	V9(2,356,0)	T41(1,361,0)	T8(4,365,0)
	T61(1,393,0)	V13(2,409,12)	T7(1,481,56)		

P: processor number; T: the task (I, J, K); I: iteration index of task T; J: expected finish time of task T as per scheduling (×10 μs); K: idle time between task T and the previous task (×10 μs).

dynamic torques which are updated and stored in the memory of the SR1. The dynamic torques result from the computation of the dynamics on modules M2 and M3. Processors P3 and P4, which interface M2 to M1, are assigned computational tasks within M2 in addition to the task of transferring the dynamic results (six 32-bit data words) from SR2 to SR1. Processors P1, . . , P10 of modules M2 and M3 (P(I), I = 1, . . . , 10) are used for testing the implementation of the Newton–Euler dynamics using schedules requiring one to ten processors. Processors P13 and P14 interface module M2 to M3 and are entirely assigned communication tasks. Processor P12 is a personal computer which is used as a software development tool and for downloading of the processor assignments. Each processor (of M1, M2 and M3) has a small monitor which allows resetting, loading of programs from the shared resource, sending system messages and synchronizing programs.

To generate schedules and codes for MRTA, the task times are evaluated according to the vector-operator times listed in Table 1. These task times are then inputted to the scheduling software and new schedules (S(M), M = 1, . . , 10) are generated. A schedule S(M) = (A(I), I = 1, . . , M) is a collection of M programs A(I) to be assigned to M processors. The implementation of a schedule S(M), M = 1, . . , 10 on the multiprocessor shown in Figure 7 consists of mapping each program A(I) of S(M) to a processor P(I).

As modules M2 and M3 support a tightly-coupled application, a message routing scheme (see above) cannot be used because of its latency. Software synchronization of the tasks is achieved by passing parameters between the tasks and is subject to the following rule. When a processor P(I), that is a member of a module (M2 or M3), completes the computation of a task T, transfer of the parameters PR(T) will be subject to the following conditions:

1 All the successor tasks of T are assigned to P(I), then PR(T) is stored in private memory of P(I),
2 At least one successor task of T is not assigned to processor P(I); two cases arise:
   - All the processors that are assigned all the successors of T are members of module M, then PR(T) is stored in the shared memory of M.
   - At least one processor P′, which is not a member of module M, is assigned some successors of T, then P(I) interrupts the communication processor (Pc), which is either P13 if M = M2 or P14 if M = M3, L seconds earlier than the completion time of T and communicates only the address of PR(T). The time L is approximately equal to the interrupt latency of Pc. In turn, processor Pc polls the version number of PR(T) and transfers PR(T) from the SR to SR′ when these parameters are available in the memory of the shared resource. This process is transparent to the

Table 4. List of tasks and their predecessors

TERM	PRED1	PRED2
V0(I)	T1(I-1)	W0(I)
T1(I)	R(I)	V(0)
V1(I)	T1(I-1)	W0(I)
V11(I)	V1(I)	W1(I)
V12(I)	T2(I-1)	V11(I)
T2(I)	R(I)	V12(I)
V2(I)	R(I)	T3(I-1)
V3(I)	T1(I)	P(I)
V4(I)	T2(I)	P(I)
T31(I)	V2(I)	V4(I)
T32(I)	T1(I)	V3(I)
T3(I)	T32(I)	T31(I)
V51(I)	T1(I)	S(I)
V5(I)	T1(I)	V51(I)
V6(I)	T2(I)	S(I)
T41(I)	T3(I)	V6(I)
T42(I)	V5(I)	T41(I)
T4(I)	M(I)	T42(I)
V7(I)	I(I)	T2(I)
V81(I)	I(I)	T1(I)
V8(I)	T1(I)	V81(I)
T5(I)	V7(I)	V8(I)

a, Forward, $I = 1, 6$

TERM	PRED1	PRED2
T61(I)	R(I+1)	T6(I+1)
T6(I)	T61(I)	T4(I)
V9(I)	P(I)	T6(I+1)
V13(I)	T7(I+1)	V9(I)
V10(I)	R(I+1)	V13(I)
V14(I)	P(I)	T4(I)
T17(I)	V14(I)	T5(I)
T7(I)	V10(I)	T71(I)
T81(I)	T7(I)	R(I)
T8(I)	T81(I)	B(I)

b, Backward, $I = 6, 1$

successor tasks of T whose processors (P') are members of M'. A processor P' can then acquire PR(T) by polling the shared resource SR' of M'.

The latter method reduces the effect of interrupt latency on the task synchronization, because the overhead on the computing processors is of the order of the time spent by P(*I*) to interrupt Pc and the latter processor to transfer PR(T).

Note that the parameter addresses, memory allocation, and routing are determined during the last stage of the scheduling process, i.e. the generated code of a task does not include the finding of successors or addresses. At the worst, task overhead consists of setting the interrupt bit of Pc and storing the results PR(T) to the appropriate memory location.

Monitoring MRTA and collecting data

Several methods[7, 13] can be used to monitor the signals of a multiprocessor. In this case, a logic analyser, DOLCH-ATLAS-9600, is used to monitor the following signals: RQ(*I*), RDY(*I*), BRDY(*I*), the port grant G(*I*) and the chip selects within each shared resource. The latching conditions are set to record only those transactions which deal with the shared resource. The transaction data are transferred to a computer and analysed by software to give the following average processor features.

Table 5. Critical path terms and priority

Critical Path Terms	Priority	Tasks Cost
V0(4)	485	4
T1(4)	481	28
V0(5)	453	4
T1(5)	449	28
V0(6)	421	4
T1(6)	417	28
V51(6)	389	42
V5(6)	347	42
T42(6)	305	8
T4(6)	297	23
V14(6)	274	42
T71(6)	232	8
T7(6)	224	4
V13(5)	220	8
V10(5)	212	20
T7(5)	192	16
V13(4)	176	8
V10(4)	168	20
T7(4)	148	16
V13(3)	132	12
V10(3)	120	24
T7(3)	96	16
V13(2)	80	4
V10(2)	76	4
T7(2)	72	16
V13(1)	56	12
V10(1)	44	24
T7(1)	20	16
T8(1)	4	4

- The schedules (S(*M*)) are run individually on the multiprocessor and the resulting LFT(*M*) now includes the overhead caused by the runtime bus contentions. Figure 8 shows the runtime performance LFT(*M*). The scheduling system generates for this example, quasi-optimum solutions for $M = 2$, 3 and 4. For $M = 5$ the error is 3.2% using Fernandez-Bussel bound. For the rest, optimum solutions were obtained. The processing power ($PP = LFT1/LFT_{min}$) was 5.2 for the generated schedules and 4.977 after their execution on MRTA. The decrease in PP is caused by processor wait time.
- Let IT be the average idle time as generated by the scheduler, i.e. 13.24% of LFT for $M = 6$, which increases significantly when more processors are used. Figure 9 shows the ratio (IT/LFT(*M*)) as a function of the number of processors. IT/LFT increases significantly for $M > LFT1/LFT_{min} = 5.2$ because of precedence relationships, i.e. no tasks could be scheduled during these intervals. The use of more than six processors for this example does not lead to any improvement in performance because the critical path of the task graph is found for $M = 6$. Figure 10 shows the average data transfer time (DTT) as generated by the scheduler, i.e. when no bus contentions are considered. DTT/LFT(*M*) decreases when $M > 6$ because DTT decreases with *M* and LFT remains constant, i.e. LFT corresponds to the critical path.
- Let WT be the average wait time spent at the multiport bus lock when a processor attempts to access the shared resource. The ratio WT/LFT(*M*) does not exceed 2.5% for $M < 7$ (Figure 11). This parameter

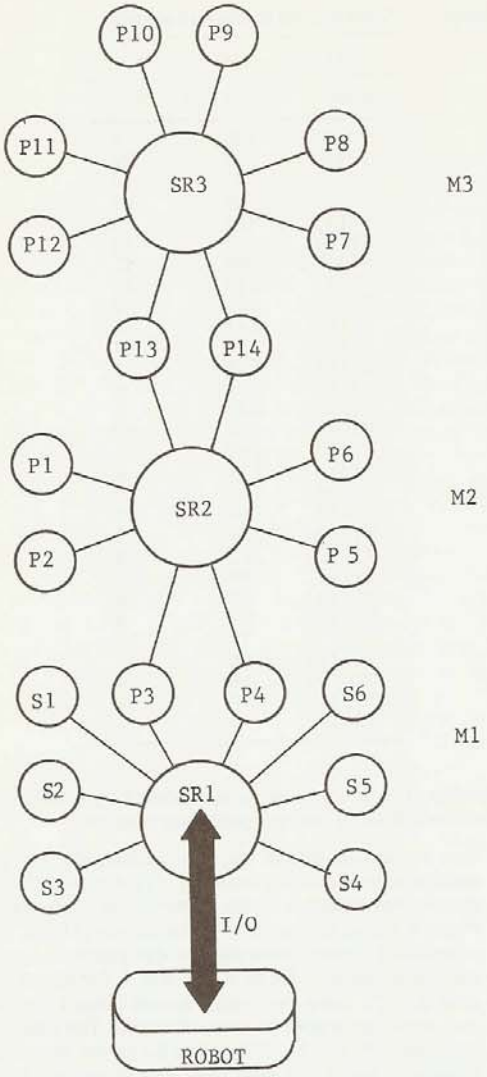


Figure 7. Implementing the dynamics using three submodules of MRTA (see text for description)

confirms expectations for the intermediate finish times of the processors: these are rarely the same because the average transfer time is only 9.55% of LFT(*M*), thus the effect of concurrent requests to the shared resource is reduced. The variations in the execution time for the fundamental operators (8% for the NDP) generates some loss of synchronization among the tasks. This causes the processors polling the version number of predecessors which in turn increases processor wait time at the multiport bus lock.

- Let PT be the average polling time spent by a processor polling the version number of predecessor tasks, i.e. those caused by unready tasks. PT, and the wait time, WT, are the direct consequence of operand variations. Figure 11 also shows PT/LFT(*M*).
- Let TT be the average overall transfer time through the shared resource. Figure 12 shows that TT does not

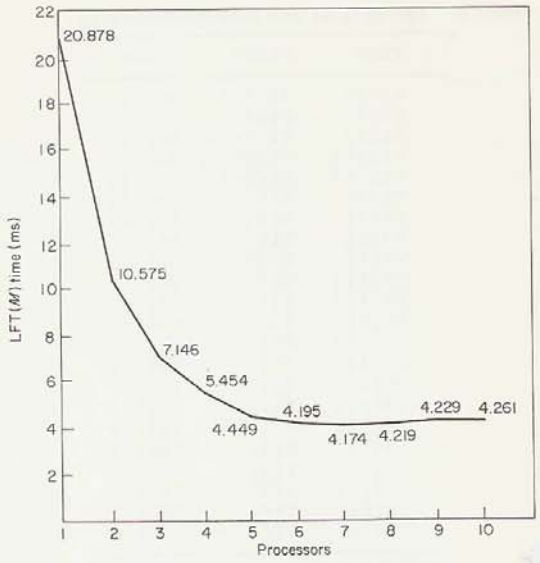


Figure 8. Longest finish time versus processors

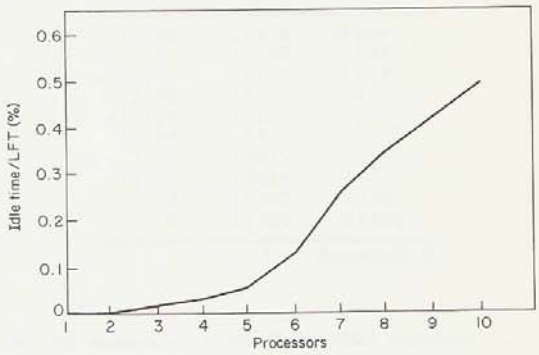


Figure 9. Average idle time IT(M)/LFT(M)

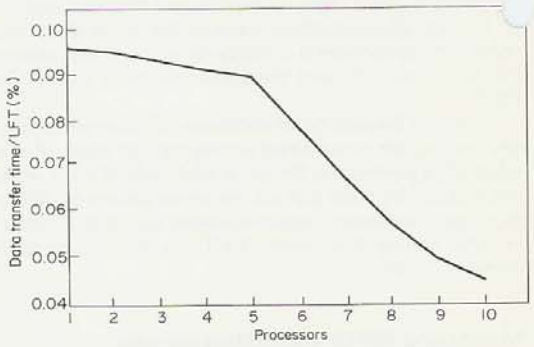


Figure 10. Average data transfer time DTT(M)/LFT(M)

exceed 13.5% of LFT(*M*). Of these, 8% are predicted by the scheduler. The upper bound of the increasing processor time, Max(TT − DTT), due to loss of synchronization, is of the order of 5.5% when compared with the processor finish times for $M = 1, \ldots, 6$.

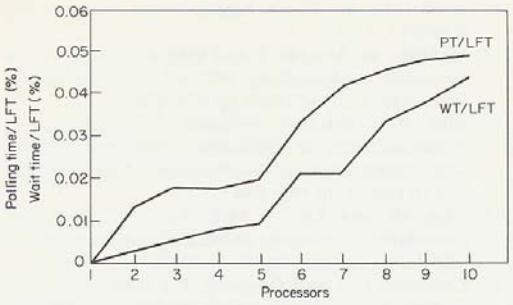


Figure 11. Average wait time WT(M)/LFT(M) and average polling time PT(M)/LFT(M)

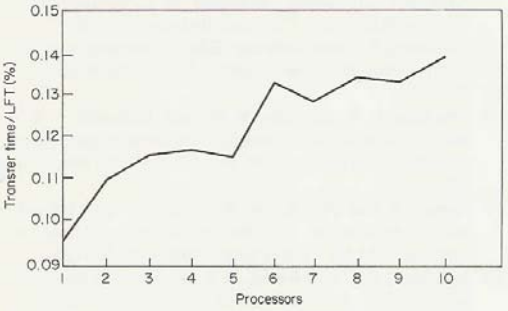


Figure 12. Average total transfer time TT(M)/LFT(M)

Analysis

The sequential structure of task programs positively dampens the variations of the processor execution times. When more than six processors are used, these variations are bounded as LFT(*M*) is caused by the calculation of CP which is equal to LFT(*M*) for $M > 5$. For $M < 6$, more stable variations were observed because LFT(*M*) involves a larger number of tasks than those of CP.

Compared with LFT(*M*), PT and WT are moderate and have a reduced effect on overall performance. This is due to two factors. First, the intermediate finish times of the processors are mainly different. There were relatively few cases where more than two processors attempted to access the shared resource. This is because the probability that a processor requests access to the shared resource is close to DTT/LFT(*M*). A dynamic priority resolution scheme (DPRS) could produce a slight improvement on LFT(*M*) because WT did not exceed 4.5% in all the cases ($M = 1, \ldots, 10$). DPRS could give the highest priority to the critical path tasks. Here, the use of semaphore is not useful because the next task will not be enabled until its predecessor data are ready and being transferred to the processor.

Second, PT is the direct cause of operand variations because a task could not start before the transfer of all the predecessor data is complete. The time spent polling the shared resource does not directly add to the finish time. The degradation is caused by the processor waiting at the multiport bus lock which is partially due to the effect of polling.

For this example, the scheduling software and the implementation on MRTA of the various schedules allowed the achievement of a finish time of 4.195 ms using six processors (P1, . . , P6 of M1) in computing the Newton-Euler dynamics. Here, the improvement in performance is due to the fact that at the vector–matrix operator level, the transfer time through the multiport bus generally takes a small fraction of the operator time in case of no bus contentions.

To study the interaction of modules M2 and M3, let C1 and C2 be the sets of experiments/schedules requiring six or less processors (M2) and seven or more processors (M2 and M3), respectively.

For experiment C1, the implementation of the retained schedule S6 on module M2 has generated a degradation in performance of 5.4% (Figure 12). Of this 3.4% is due to the loss of synchronization, which is caused by operand variations, and 2% is due to concurrent access to the multiport bus switch of M2. In this example, the required bandwidth was 28% of the available bandwidth. Here, software synchronization among the tasks is not a bottleneck because of the optimized timing of the processor access to the shared memory (see above) and the moderate ratio of the task transfer time to task execution time (see above).

The experiments of C2 showed a linear decrease in performance compared with those of C1. This clearly arises from the average increase in functions WT, PT, and TT for $M = 1, \ldots, 10$ (Figures 11 and 12). This suggests that tightly coupled applications can be distributed on two MRTA modules without dramatic performance degradation. Further study of intermodule communications with frequent data passing would be an interesting issue for MRTA.

This example and others[23] of robot control have proved the usefulness of this system for developing parallel algorithms with higher processing power than that obtained using enumerative methods.

CONCLUSION

In a robot controller, lower levels usually perform regular calculations with intensive interaction, while higher levels evaluate refined global decisions with relatively slower interaction. At all levels, parallelism can be useful to speed up the process control.

The MRTA multiprocessor was designed and implemented to allow the building of suitable interconnection topologies by using smaller multiprocessor modules. A module can support the implementation of a tightly coupled calculation. Thus multiple or hierarchical structures of the controller can then be mapped to the multiple module architecture of MRTA. As processors of such a system need not be fully interconnected, modules can be loosely or tightly connected using processors. The MRTA modular architecture is reconfigurable and expandable at the levels of the module interconnection topology and the shared resource. The development of parallel algorithms for robot control consists of decomposing the original model of the process into tasks and scheduling their computation such that the minimum finish time is achieved. This problem is very difficult to solve in the general case: a graph-structure for the tasks and an arbitrary number of processors. No optimization algorithms with polynomial time are known to find the optimum solution. One approach to this problem is to use an efficient approximation method whose perfor-

mance is guaranteed in the worst case. The method is constraint-free regarding the problem size and the number of processors. The proposed software admits high-level formulation of control problems, performs task generation, executes list scheduling and generates executable processor assignments for MRTA modules.

Based on the Newton–Euler formulation of dynamics, an example of robot control has been investigated and implemented using three MRTA modules. The developed software/hardware enabled partitioning of the problem into tasks and the study of the implication of this process on the resulting transfer time. The effects on performance degradation caused by loss of synchronization were analysed together with the overall application performance.

This methodology is useful for developing parallel algorithms and evaluating their performance within the area of realtime robot control.

REFERENCES

1 **Luh, J Y S** 'An anatomy of industrial robots and their controls' *IEEE Trans. Automatic Control* Vol AC-28 (February 1983) pp 133–153

2 **Bourbakis, N G** 'A quadtree multimicroprocessor architecture for robot vision systems' *J. Microprocess. Microprogram.* Vol 16 (1985) pp 267–272

3 **Inoue, H *et al.*** 'Design and implementation of high level robot language' *Proc. 11th Int. Symp. on Industrial Robots* (1981) pp 675–682

4 **Luh, J Y S and Lin, C S** 'Scheduling of parallel computation for a computer-controlled mechanical manipulator' *IEEE Trans. Syst. Man. Cyber.* Vol SMC-12 No 2 (March 1982)

5 **Hwang, K and Briggs, F A (Eds.)** *Computer Architecture and Parallel Processing* McGraw-Hill, New York, NY, USA (1984)

6 **Barron, I *et al.*** 'Transputer does 5 or more MIPS even when not used in parallel' *Electronics* (November 1983)

7 **Ghosal, D and Patnaik, L M** 'SHAMP: an experimental shared memory multimicroprocessor system for performance evaluation of parallel algorithms' *Microprocess. Microprogram.* Vol 19 (1987) pp 179–192

8 **Mudge, T N, Hayes, J P and Winsor, D C** 'Multiple bus architectures' *Comput. J.* (June 1987) pp 42–48

9 **Debaere, E H and Van Campenhout, J M** 'A shared-memory Modula-2 multiprocessor for real time control applications' *Microprocess. Microprogram.* Vol 18 (1986) pp 213–220

10 **Kasahara, H and Narita, S** 'Parallel processing of robot-arm control computation on a multimicroprocessor system' *IEEE J. Robotics and Automation* Vol RA-1 No 2 (June 1985)

11 **Intel** *IAPX 86 User's Manual* Intel, Santa Clara, CA95051, USA

12 **Handler, W, Maehle, E and Wirl, K** 'DIRMU multiprocessor configurations' *14th Int. Conf. on Parallel Processing* (August 1985) pp 652–656

13 **Klar, R** 'VLSI-based monitoring of inter-process-communication in multi-microcomputer systems with shared memory' *Microprocess. Microprogram.* Vol 18 (1986) pp 195–204

14 **Chu, W and Lan, L M-T** 'Task allocation and precedence relations for distributed real-time systems' *IEEE Trans. Comput.* Vol C-36 No 6 (June 1987)

15 **Coffman, E G and Denning, P J (Eds.)** *Operating Systems Theory* Prentice-Hall, Englewood Cliffs, NJ, USA (1973)

16 **Ramamoorthy, C V, Chandy, K M and Gonzalez, M J** 'Optimal scheduling strategies in a multiprocessor system' *IEEE Trans. Comput.* (February 1972)

17 **Coffman, E G and Graham (Eds.)** *Computer and job-shop scheduling theory* John Wiley, Chichester, UK (1976)

18 **Thomas, L A, Chandy, K M and Dickson, J R** 'A comparison of list schedules for parallel processing systems' *Commun. ACM* Vol 17 No 12 (December 1974)

19 **Ayyad, A and Wilkinson, B** 'Multiprocessor scheme with application to macrodataflow' *Microproc. Microsyst.* Vol 11 No 5 (June 1987) pp 255–263

20 **Nelson, J C C and Refai, M K** 'Design of a hardware arbiter for multiprocessor systems' *Microproc. Microsyst.* Vol 8 (1984) pp 21–24

21 **Hollerback, J M** 'A recursive Lagrangian formulation of manipulator dynamics and comparative study of dynamics formulation complexity' *IEEE Trans. Syst. Man. Cyber.* Vol SMC-10 No 11 (November 1980)

22 **Nigam, R and Lee, C S** 'A multiprocessor-based controller for the control of mechanical manipulators' *IEEE J. Robotics and Automation* Vol RA-1 No 4 (December 1985)

23 **Al-Mouhamed, M A and Bubshait, F A** 'A comparative study of scheduling large number of iterative tasks' *Fourth Int. Symp. on Computer and Information Sciences* Cesme, Turkey (October 1989)

Mayez Al-Mouhamed is an assistant professor at KFUPM. He received the Maitrise Es Sciences, Doctorat De Troisieme Cycle, and Doctorat Es Sciences degrees in electrical engineering from the University of Paris in 1975, 1977 and 1982, respectively. From 1975 to 1983, he worked on the design of the French Robot System (MA-23) at the Nuclear Research Center, Saclay, France. During this period, he registered two European Patents in the area of robotics. His research interests include robot system design, computer architecture and sensory systems.