# Real-Time Communications in a Computer-Controlled Workcell

Kang G. Shin, *Senior Member, IEEE*

***Abstract*—A computer-integrated manufacturing (CIM) system is composed of several workcells, each of which contains robots, NC machines, sensors, and a transport mechanism. Each CIM workcell is controlled by multiple processors interconnected by an intracell bus, but the individual cells are interconnected by an arbitrary network. This paper considers a communication subsystem that is designed to support real-time control and coordination of devices in each CIM cell. The concept of a *poll number* is proposed to control the access to the intracell bus. The bus access mechanism with the poll number is intended to minimize the probability of real-time messages missing their deadlines. When a CIM task generates a time-constrained message, a poll number is computed for this message according to the message's deadline and the task's priority. When the intracell bus is free, the various tasks at a workcell that desire to use the bus write the poll number onto the bus and read it back, one bit at a time, starting from the most significant bit. If at any time the bit read back is different from the bit written, then the corresponding task drops out of the contention for the bus. Use of a poll number provides not only for decentralized control of the intracell bus, but also a high degree of flexibility in scheduling messages. The performance of the bus access mechanism with a poll number is analyzed and compared with that of a token bus, which is widely used in CIM systems such as MAP networks. The probability of a real-time message missing its deadline in a token bus is found to be much higher than that of the proposed mechanism.**

***Index Terms*—Computer-integrated manufacturing (CIM), real-time tasks and messages, deadlines, message scheduling, token bus, poll number, polled bus.**

## I. Introduction

THE factory of the future, or the factory with a future, is based on the notion of computer-integrated manufacturing (CIM), whose ultimate goal is to integrate and control manufacturing processes, material inventory, sales and purchases, administration and accounting, and engineering design information into a single, closed-loop and interactive control system. Essential to CIM is the computer communication network over which the information necessary for process interaction and coordination, status checking, and plant monitoring/control will be exchanged.

Although the technology of computer networking is quite advanced, its specific application to CIM has not been well addressed. A large body of analytic research exists in the general area of network control and communications, focusing on modeling and performance evaluation as well as on designing flexible systems for accommodating future growth. However, it seldom addresses such CIM needs as the compatibility between equipment manufactured by different vendors, and real-time communications between workcell devices [1]. Using the token bus (IEEE 802.4), the seven-layer broadband Manufacturing Automation Protocol (MAP) is proposed by GM and other companies to provide temporal ordering consistency as well as to eliminate incompatibility between pieces of equipment [2], [3]. However, this seven-layer MAP is usually too slow to handle real-time communications in a workcell, thus prompting the introduction of an additional type of node called the MINIMAP. The MINIMAP employs only the first two layers of the MAP and combines the remaining five layers of the MAP into a single layer. Although the communication delay is expected to decrease by a reduction of the number of layers, the real-time performance of MINIMAP is still unknown. In fact, as we shall see, the source of MAP's long communication delay is not only the number of layers but also the way the token bus works. The main goal of this paper is to address the latter issue. We will do this by proposing a new bus access mechanism using a *poll number*, which is computed based on the deadlines of the corresponding message and task. The bus equipped with this access mechanism will henceforth be called the *polled bus.*

Manuscript received January 9, 1989; revised January 13, 1990. This work was supported by the National Science Foundation under Grant DMC-8721492.

The author is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122.

IEEE Log Number 9041450.

A CIM system usually consists of several workcells, each of which contains robots, NC machines, sensors, and a transport mechanism. The devices in a workcell need to be coordinated and controlled to perform such collective functions as assembly and material handling. These workcell devices are usually controlled by multiple processors interconnected by an intracell bus, but the individual cells are assumed to be interconnected by an arbitrary network. Since, for example, the cars on an assembly line move at a fixed speed regardless of whether robots are ready or not, it is essential to know the worst-case communication delay in advance. So, one cannot use a protocol for CIM intracell communications unless it guarantees a bounded worst-case delay. For example, Ethernet [4] cannot be used for intracell communications because of the possibility of its unbounded delay. In general, all CSMA/CD protocols are not applicable to intracell communications owing to their inability to guarantee bounded communication delays. See [5] for an in-depth analysis of the delay characteristics of CSMA/CD networks and their use to determine the suitability of CSMA/CD with random service order and FCFS policies for different applications that require probabilistic (instead of absolute) delay bounds. On the other hand, the token bus protocol guarantees a bounded communication delay, though the delay could be quite large. (This is one of the reasons, along with its ability to provide temporal ordering consistency, why the token bus was chosen for MAP.)

Communication delays longer than a specified value or *deadline* could lead to a collision between two robots, or disrupt the assembly line operations, resulting in loss of production time and product quality. Note that the timely completion of tasks controlling cell devices strongly depends on the communication delay between the cooperating tasks. For example, in visual servoing applications, the control task can complete execution only after receiving the necessary information on concerned objects from the vision processing task. To characterize the

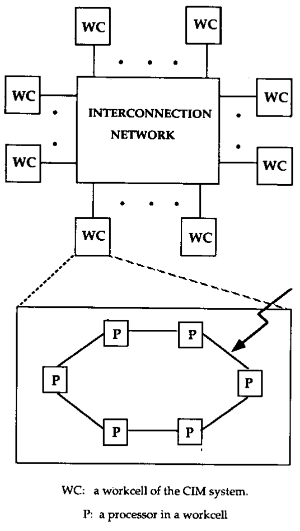

WC
INTERCONNECTION NETWORK
Polled Bus
P
WC: a workcell of the CIM system.
P: a processor in a workcell

Fig. 1. A model of a distributed CIM system.

system's ability of completing time-critical tasks before their deadlines, the *probability of dynamic failure* was defined as the probability of one or more tasks missing deadlines [6]. As mentioned above, this probability is strongly dependent on the mechanisms used to implement intertask communications. The work described in this paper is primarily targeted at minimizing the probability of dynamic failure by speeding up the intertask communication within a workcell.

This paper is organized as follows. The next section describes the CIM communication subsystem, compares it with general-purpose networking approaches, and introduces the polled bus as an intracell communication mechanism. Operation of the polled bus and design of the poll number, along with the benefits of using the poll number, are discussed in Section III. In Section IV, the performances of the polled bus and the token bus are analyzed and compared. The paper concludes with Section V.

## II. Putting Things in Perspective

Due to the inherent nature of a CIM system, it is natural to control the CIM system with a distributed computing system consisting of a two-level hierarchy: *workcell* and *network* levels. Each workcell is controlled by several processors connected via a simple, high-speed bus, whereas these workcells are interconnected via an arbitrary network. Each processor in a cell executes one or more tasks, realizing some useful, probably time-constrained, functions, such as controlling a robot or interpreting sensor data. It is often necessary that the tasks, within a cell as well on different cells, exchange information to realize the overall functions of the CIM system.

Since one or more devices in the same workcell may have to work together to accomplish a common goal (e.g., assembly of parts), the tasks that control these devices typically have to meet stringent timing constraints. All the processors that execute these tasks within a workcell are placed on a single board or chassis along with a communication processor (CP). The CP is responsible for intertask communications within a cell as well as across cell boundaries.

The CIM system model is depicted in Fig. 1. The model is similar to the one described in [7], where the processors within a cell are connected by a token bus [8]. The main difference between a token bus and the polled bus lies in their access protocols.

Both the token bus and the polled bus may be categorized as *broadcast buses*, each of which is defined as a bus structure wherein a processor need not be aware of the other users of the bus. We do not restrict the type of interconnection between the various processors as long as it has broadcasting capability. Communication across cell boundaries does not usually have any real-time constraints, e.g., exchange of inventory or sales information. Typically, the cells are connected by a token-ring local-area network. The tasks communicate with each other via message passing [9]. A seven-layer protocol such as the MAP [3] may be used for communication *across* cell boundaries, but, as mentioned earlier, such a protocol is usually too slow to be used for communications within a workcell. One of the CP's functions is to provide the protocol support for implementing communications between any two tasks, which may either be in the same cell or in different cells.

The tasks within a workcell are responsible for controlling real-time devices such as robots and sensors. The tasks for controlling and sensing these devices are inherently periodic; for example, the task for closing a digital servo control loop may be executed once every 100 ms, and the task for sensing and analyzing parts on a conveyor belt may have to be executed once every 0.5 s for visual servoing. Aperiodic tasks, albeit infrequent, also exist within a workcell, e.g., and operator's commands in response to abnormal circumstances. Periodic tasks are "based load" whereas periodic tasks are "random disturbances." The main intent of this paper is to deal with the "base load" or periodic tasks; treatment of aperiodic tasks is usually formulated as a dynamic load-sharing problem [10].

The bus access mechanism used in the token bus of [11] is the token ring protocol, where a single token goes around the ring. Any processor on the ring desirous of using the bus should capture the token and release it upon completion of its service. This mechanism distributes access in a round robin manner, wherein higher priority task (processors) might be forced to wait longer than necessary to procure the bus while the lower priority tasks are accessing the bus. This problem may partially be solved by providing multiple priorities within the token passing bus scheme. In the token passing bus method, the *class of service* mechanism can be used to provide prioritized (4 levels) access [12]. The token bus access mechanism belongs to a class of Controlled Demand-Adaptive Multiple Access Protocols [13]. Priority access schemes using CSMA/CD for multiple priority message classes have also been proposed and analyzed [14]. According to [13], these schemes belong to the class of Contention-Based Multiple Access Protocols.

Controlled Demand-Adaptive Multiple Access Protocols can be broadly divided into token-passing and reservation schemes [13]. The token bus is an example of a token-passing scheme. The reservation scheme involves a reservation period that is divided into slots, wherein all stations (processors) that have messages to transmit post their reservation by transmitting a burst of noise during their assigned slot. After the completion of the reservation period, a station is selected based on a predetermined scheme known to all stations. The reservation scheduling protocol (RSP) proposed in [15] falls into this reservation scheme category. The RSP scheme requires a reservation period, during which the highest priority message is selected (irrespective of the station from which it originated), followed by a scheduling period, during which the station is selected based on some scheme (e.g., round robin). However, there are two drawbacks associated with the RSP protocol. First, it requires two sequential steps for reservation and scheduling, which induce a longer delay in message delivery than using only one short combined step. Second, the scheduling period and the selection schemes of the RSP are sensitive to the number of stations using the bus.

To remedy the drawbacks of the RSP and a token bus (e.g., MAP and MINIMAP), and thus, to minimize the probability of a CIM task missing its deadline, we propose a new bus access mechanism, which is somewhat similar to the one used in FTMP [16]. Unlike the RSP protocol [15], our scheme combines the reservation and scheduling periods into a single *polling round*. It ensures that at the end of the polling round, only one station (processor) will have control of the bus.

In our scheme, each processor computes a *poll number* that is determined by various factors, including the corresponding message's deadline and user-assigned priorities. When a task generates a message service request, it waits for the bus to complete servicing the present request and enters a polling round together with the other processors wanting to use the bus. After a fixed amount of time called the *polling time*, the processor with the highest poll number is guaranteed to get control of the bus. This proves to be superior to the token bus since it gives higher priority to tasks with closer deadlines. This scheme is better than the RSP, since the polling round is not dependent on the number of processors. In our scheme, each processor need not be aware of the other processors' priorities, nor of the number of such processors contending for the bus.

Because of the nature of the bus, we refer to the intracell bus as a *polled bus* in contrast to the token bus of [11]. The proposed bus access mechanism also provides a decentralized bus access mechanism with predictable and optimum performance parameters.

## III. The Polled Bus Access Mechanism

Communications within a cell must be completed in a timely fashion so as to meet the real-time requirements of the various tasks that are responsible for controlling and coordinating the workcell devices. The design of the interconnection mechanism for processors within the same cell has to satisfy these real-time requirements. The interconnection mechanism (bus) may be controlled by a dedicated processor that is central to the cell. This processor could then decide on the allocation of the bus to the various tasks so that the probability of missing deadlines is minimized. However, the failure of the central control processor would paralyze the communications within a cell leading to a potentially disastrous situation. Hence, we propose a bus access mechanism that provides for decentralized control of the bus.

As mentioned earlier, the processor in a cell are interconnected by a broadcast bus, which enables the processors to read from and write into the bus without being aware of the presence of other processors. Typical examples are time-shared unibus, token ring, token bus, etc. The software executing on each processor may be partitioned into device control and interface software. Among the important functions of the interface software is bus access. The software implements the decentralized bus access algorithm, which will be described in Subsection III-A.

In [11], the various processors within a workcell are interconnected by a token bus, where a token circulates around the processors. The processor that needs to send a message using the bus should capture the token to control the bus and release the token for circulation as soon as it completes the message sending. The token travels to the logically adjacent processor on the bus, as in the case of a token ring.

Consider the four processors $P_1$, $P_2$, $P_3$, and $P_4$ connected to the token bus. Assume that the task priorities are such that $Pr_1 \leq Pr_2 \leq Pr_3 \leq Pr_4$, where $Pr_i$ represents the priority of the task executing on processor $i$. This means that $P_4$ executes the most critical task(s). If $P_4$ sends a message and then relinquishes the bus (token) to the next processor downstream (i.e., $P_1$), then it has to wait the whole round before sending the next message, if any. The best-case scenario is that none of $P_1$, $P_2$, and $P_3$ have any message to send. The worst case occurs when the other three processors have messages to send. In this case, the task executing on $P_4$ may get delayed and may even miss the deadline, which might prove catastrophic if it has to execute a critical task. From the above discussion, it is easy to see that the task with the highest priority among the tasks competing for the bus should be given control of the bus. These priorities may be user-specified, or in the absence of any user specification, may be assumed to be the time left for the lapse of the deadline. In our proposed mechanism the above scenario will not occur since the task whose message has the earliest deadline and/or the highest priority will always get the bus before any other task.

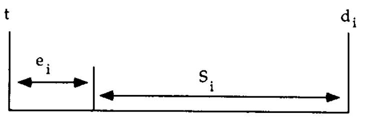
### A. Bus Acquisition Algorithm

A high-speed bus interconnects all the processors of a cell. It is assumed that the processors can detect if the bus is busy. This may be accomplished by the use of a bus busy line. The logical structure of the bus is circular. Whenever the bus is in use (for sending messages), a line (busy line) is set high. All the processors connected to the bus can sample this line.

When a processor needs to use the bus, it first samples the busy line. If the bus is busy, the processor busy-waits for the bus. As soon as the bus is free, the processor attempts to acquire the bus by initiating the acquisition process. Any number of processors might attempt to acquire the bus simultaneously. Each processor computes a number (which should be unique) called the *poll number*, which consists of a finite number of bits, say, $m$ bits. These bits are divided into several fields with each field having a special physical significance. Determination of the exact structure of the poll number and its design will be discussed in Subsection III-B. This number is guaranteed to be unique to prevent a tie, i.e., no two processors can produce the same poll number. This poll number is so designed that the task whose message has the earliest deadline will have the largest poll number. The bus performs a wired-OR operation on all the signals impinging on it from the various processors. A processor competing for the bus writes the poll number to the bus, one bit at a time, starting with the most significant bit. After writing each bit, it waits for a finite period and samples the bus in order to propagate and stabilize the bit written on the bus. This waiting period is a function of the physical length of the bus. If the value read by the processor is different from the value it wrote into the bus, it drops out of contention for the bus. This situation will occur only when a processor with a larger poll number is contending for the bus. After $m$ such rounds, the processor with the highest poll number has sole control of the bus.

### B. Design of the Poll Number

The poll number is computed by every processor that needs to use the bus. The design of the poll number acutely affects the performance of the system. The algorithm does not require any



t : present time (bus needed).

$e_i$ : execution time (estimated).

$S_i$ : slack time (time left for deadline).

$d_i$ : deadline.

Fig. 2. Defining the "slack time."

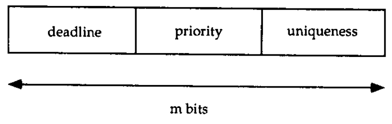


Fig. 3. Field structure of the poll number.

state exchange between processors, thereby obviating the need for maintaining the global system state at each cell.

The main issues in the design of the poll number are:

1) the number of bits $m$ used to encode the poll number and how to ensure its uniqueness without sacrificing other information and
2) the significance and the ordering of the various fields of the poll number so as to minimize the number of messages missing their deadlines.

First, we shall determine the fields of the poll number and their relative ordering. A field is a contiguous collection of bits that has special significance. A field, called the ***uniqueness*** field, assigns a unique identification number to each of the processors in a cell so that no two poll numbers will be identical in any situation. Another field, called the ***deadline*** field, is necessary to represent slack time (the time left until the deadline) or some encoding of it. (See Fig. 2 for the definition of slack time.) The user-defined priorities of the tasks are defined in the *priority* field. It is easily seen that the uniqueness field design depends on the number of processors and the priority field is related to the tasks that reside in these processors.

The ordering of these fields depends on the system design objectives. In our design we would like to minimize the number of messages missing their deadlines. Hence, the most significant field is the deadline field in which each message's deadline is encoded to be *inversely* proportional to its actual deadline. The next significant field is the priority field. In a situation where two competing tasks have the same deadline, the task with the greater user/system assigned priority gets the bus. The least significant is the uniqueness field, which comes into play only when competing processors have identical deadline and priority fields. In order to avoid biasing the bus access toward any one processor on account of the uniqueness number, the processor identification numbers that form the uniqueness field component can be assigned in a round robin fashion. The ordering of the three fields of the poll number is shown in Fig. 3.

The number of bits in the poll number $m$ will now be determined. We shall first do a simple analysis and obtain an expression for $m$. Then we shall show with the help of typical numerical values that the value obtained for $m$ is far in excess of what is actually required. A simple encoding is used to obtain the poll number, in particular, the deadline field. (Other encodings, which might result in a better performance, are possible but will not be considered here.)

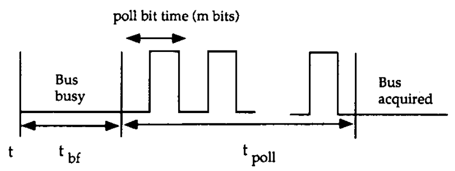


Fig. 4. A typical bus access cycle in a poll bus.

Let $N_{\text{proc}}$ be the number of processors in a cell and $N_{\text{tasks}}$ be the number of tasks resident at the cell. (If only a single task resides at a single processor, then $N_{\text{proc}} = N_{\text{tasks}}$.) The number of bits needed in the uniqueness field is $\log_2 N_{\text{proc}}$, and the number of bits needed for the priority field is $\log_2 N_{\text{tasks}}$. Unlike the uniqueness and priority fields, determination of the maximum number of bits needed to represent the deadline field is not so straightforward.

Consider Fig. 4, which shows a bus access cycle.

At time $t$, the processor (or task executing on that processor) generates a message and thus requires the bus to send it. The time interval between the generation of messages is assumed to be exponentially distributed. The processor waits for a time $t_{bf}$ for the bus to become free. Then it enters the polling round, irrespective of whether it has any competitors or not. The polling round consists of $m$ poll bit times, where each ***poll bit time*** consists of a write, an interval to stabilize the bit written on the bus, and a read by the processors involved. Let the time taken for each polling round be $t_{\text{poll}}$, which is a linear function of $m$. It is necessary that all the processors in the same cell be tightly synchronized, which can be accomplished via hardware clock synchronization similar to the one in [17]. After a processor gains control of the bus, it sets the bus busy line high.

A polling round is necessary before every message is sent. It will be shown later that the overhead caused by polling is insignificant. Let $P_{\max}$ denote the maximum period of all the periodic tasks[1] resident at a cell. The deadline field will have to be large enough to represent $P_{\max}$, since it is the largest possible deadline, though actual message deadlines are usually much smaller than $P_{\max}$.

The resolution of the deadline field has to be determined. If all system time is counted in number of clock cycles, then the minimum resolution necessary is $t_{\text{poll}}$ cycles—the time taken for the polling round in clock cycles. A finer resolution than this will not serve any purpose, since it takes at least $t_{\text{poll}}$ cycles for the processor to get the bus. At least one bit of the poll number should change every $t_{\text{poll}}$ cycle. For example, in the Fault-Tolerant Multiprocessor (FTMP) [16], the polling round takes 9-bit cycles. In this case, the least significant bit (LSB) of the deadline field would represent the time period of 9-bit cycles or more

The maximum number of bits needed in the deadline field is given by $\log_2(P_{\max}/t_{\text{poll}})$. The polling time is a linear function of the number of poll bit times in the polling round, which is the same as the number $m$ of bits in the poll number. Thus $t_{\text{poll}} = cm$, where $c$ is some constant. The number of bits ($m$) in the pole number is given by

$$m = \log_2 N_{\text{proc}} + \log_2 N_{\text{tasks}} + \log_2(P_{\max}/t_{\text{poll}})$$
$$= \log_2(N_{\text{proc}} N_{\text{tasks}} P_{\max}/cm). \quad (1)$$

[1] If some of the tasks are aperiodic, this would represent the maximum interactuation delay.

| | | |
|---|---|---|
| d2 d1 d0 | p2 p1 p0 | u2 u1 u0 |

| DEADLINE (ms) | CODING | | | d2 | d1 | d0 |
|---|---|---|---|---|---|---|
| 0 - 20 | 0 | 0 | 0 | 1 | 1 | 1 |
| 20 - 40 | 0 | 0 | 1 | 1 | 1 | 0 |
| 40 - 60 | 0 | 1 | 0 | 1 | 0 | 1 |
| 60 - 80 | 0 | 1 | 1 | 1 | 0 | 0 |
| 80 - 100 | 1 | 0 | 0 | 0 | 1 | 1 |
| 100 - 200 | 1 | 0 | 1 | 0 | 1 | 0 |
| 200 - 300 | 1 | 1 | 0 | 0 | 0 | 1 |
| 300 - - | 1 | 1 | 1 | 0 | 0 | 0 |

Fig. 5. Possible design of a poll number.

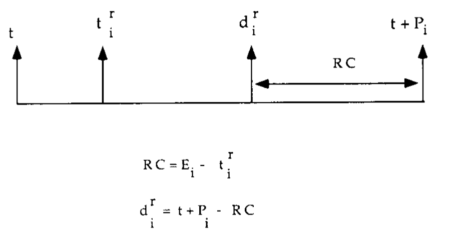


Fig. 6. Estimating the deadline of a message.

Equation (1) reduces to the form $m = K2^m$, which can then be solved to obtain the number of bits in the poll number.

As an example, again, consider the FTMP [16]. It has a clock rate of 8 MHz (125 ns). The real-time workload on FTMP consists of three task classes of periods 40 ms (25 Hz), 80 ms (12.5 Hz), and 320 ms (3.125 Hz). The poll bit time for FTMP is 1 $\mu$s. If we use (1) to compute the number of bits in the poll number, we have $t_{\text{poll}} = 15$ $\mu$s and $P_{\max} = 320$ ms. Approximately, we need 16 bits for the deadline field.

The number obtained by using the above expression for $m$ is extremely conservative in nature. The design of a poll number with a 3-bit deadline field, 3-bit priority field, and 3-bit uniqueness field is shown in Fig. 5. The resolution in this case is the minimum task period, which is 40 ms.

## C. Estimation of Message Deadlines

Knowledge of the deadline for a message generated by a task is essential for determining the deadline field of a poll number. We propose a method to determine the deadline for a message generated by a real-time task. For the clarity of presentation, it is assumed that all the tasks are initiated at the beginning of their periods, and each task executes on a single processor.

The period of a task $T_i$ is depicted in Fig. 6. Since $T_i$ has been initiated at time $t$, the next initiation of the task will be at $t + P_i$, where $P_i$ is the period of $T_i$. During the course of its execution the task will generate messages. Upon generation of a message, the task will be blocked at least until the message is put on the bus. It may or may not wait for the reply, before it resumes execution, depending on whether it executed a blocking send or a nonblocking send [18]. The estimated completion time for task $T_i$ (not including the delay while waiting for the bus) is given by $E_i$,which includes the time spent in blocking while waiting for a reply. The $r$th message of this task is generated at $t_i^r$, when it has completed $c_i^r$ units of execution time. In addition to the local clock, each processor keeps track (using a simple counter) of the execution time of its task, which does not include the time spent by the task while waiting for the bus. At $t_i^r$, $T_i$ has generated its $r$th message and still needs the *residual computation time* $RC = E_i - c_i^r$ to complete execution. The deadline for the $r$th message is $d_i^r = (t + P_i) - RC$, as the task has to send its message by $d_i^r$ if it is to complete the task by $t + P_i$.

$E_i$ and $P_i$ are known in advance; as soon as the processor starts executing the task at time $t$, $t + P_i$ can be determined. At any time $RC$ can be determined by subtracting the reading of the execution time from $E_i$. The value of deadline computed for the $r$th message ($d_i^r$) is then complemented,[2] resolved into coarser units, and loaded into the deadline field of the poll number. The priority and the uniqueness fields may be loaded at the same time.

## D. Benefits of the Poll Number Approach

The bus access mechanism with the poll number makes message scheduling flexible. A number of schemes can be implemented by varying the order and significance of the various fields making up the poll number. For example, we can implement earliest deadline scheduling by loading the task deadline into the priority field. Priority-driven scheduling can be implemented by making the priority field the most significant field. Likewise, deadline-driven scheduling may be implemented by making the deadline field the most significant field.

We shall discuss the modifications needed to make the poll number based access mechanism compatible with the earliest deadline scheduling scheme. The deadline field, as before, represents the message deadline. The priority field has been modified to hold the particular task invocation's deadline.[3] Let $TD_i = t_i + P_i$ and $TD_j = t_j + P_j$ denote the deadlines of $T_i$ and $T_j$, respectively.[4] Let the uniqueness field of the poll number for $T_i$ be $U_i$, and that for $T_j$ be $U_j$. (By definition of the uniqueness field of the poll number, $U_i \neq U_j$ $\forall i \neq j$.) With the above design, if a message from a task $T_i$, with a deadline $d_i$, competed for the bus with message from another task $T_j$, with a deadline $d_j$, then $T_i$ would get the bus if and only if

1) $d_i \leq d_j$ or
2) $d_i = d_j$ and $TD_i \leq TD_j$ or
3) $d_i = d_j$ and $TD_i = TD_j$ and $U_i < U_j$.

It is easy to see that the above scheme is compatible with earliest deadline scheduling, which will schedule the message with the earliest deadline and, in the case of a tie, will give preference to the task with the closer deadline.

It is also possible to make the polled bus access mechanism compatible with a priority-driven scheme, where the various tasks have dynamically assigned priorities. In this scheme, the priorities of the tasks are assigned so as to satisfy some criterion, such as enabling a task to dispatch its message before its

[2] This is because the processor with the highest poll number wins the polling round.

[3] If task $T_i$ had been invoked at time $t_i$, then this field would hold $t_i + P_i$.

[4] These numbers are loaded into the priority fields of their respective poll numbers.

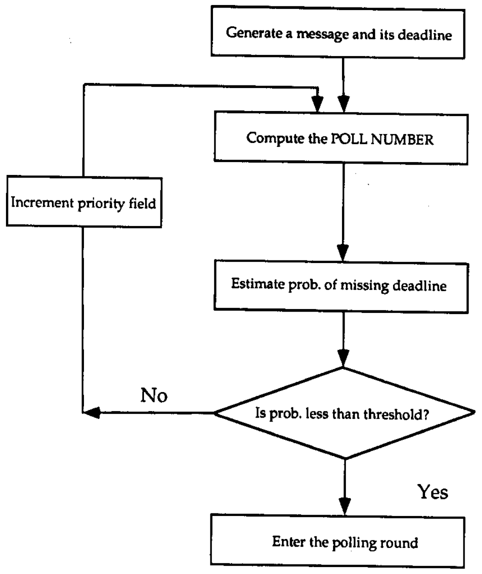

Generate a message and its deadline
Compute the POLL NUMBER
Increment priority field
Estimate prob. of missing deadline
Is prob. less than threshold?
No
Yes
Enter the polling round

Fig. 7. Dynamic priority-based scheme using a poll number.

deadline. The proposed scheme is outlined in Fig. 7. When a task needs to use the bus, it generates a poll number based upon its priority at the moment and the deadline of its message. The poll number is designed such that the priority field is more significant than the deadline field. Based upon this poll number, the task estimates the probability of the message missing its deadline. The probability of a message missing its deadline may be computed using (5) and (6) in Section IV. In order to accomplish this, the task has to be aware of the service rates and the message generation rates of the other tasks at the cell. A task has to be updated whenever the configuration of the cell is changed.

After generating a message, the task computes a poll number, and estimates the probability of the message missing its deadline. If the probability is higher than a predetermined threshold, the task recomputes its poll number after incrementing the priority field. The priority field is incremented until the probability of the message missing its deadline falls below the threshold. As soon as the probability falls below a threshold, the task then enters the polling round for the bus.

If all the tasks at the cell increase their priorities to minimize their probabilities of missing deadlines, then it is highly likely that all tasks might end up having the same probability. In this case the scheme reduces to deadline-driven scheduling. If the priority fields of the poll number are all equal, then the deadline fields will determine the task that will get the bus, as we have made the priority field the most significant field. This situation may be avoided by allowing only selected tasks to modify their priorities.

## IV. Performance Analysis

Since the token bus is widely used for CIM communications such as MAP, we shall compare the performance of the token bus with that of the polled bus. As stated before, our main objective is to minimize the number of messages missing their deadlines and to service each communication request as quickly as possible. The probability of real-time messages missing deadlines will be computed for both the token bus and the polled bus under a commonly used assumption that message arrivals at each node/processor follow a Poisson process, i.e., Markovian arrivals. Such a computation is known to be intractable without the assumption of Markovian arrivals.

### *A. The Token Bus*

A token bus is a single bus to which a number of processors are connected, and its access protocol is very similar to that of a token ring. A token is passed around processors on the bus and any processor that desires to use the bus has to capture the token first. After using the bus, the processor hands the token over to the logically adjacent processor on the bus.

Only a single token is allowed to exist on the bus at any time when the bus is not being used by any processor. The token is regenerated when all the bits of the packet sent out by a processor are received by the same processor again. This is possible as the data on the bus is seen by all the processors on the token bus, which is an example of a broadcast bus. This acts as an acknowledgment for the sending processor. We assume a bit serial bus with a typical bit rate of 10 Mbit/s or higher. There is a bus busy line that the processors can sample in order to check whether the bus is in use. We also assume that only a single task executes on each processor.

The following three main events will occur when sending a message:

1) A task generates a message.
2) The processor executing the task waits to get control of the bus.
3) The message is transmitted via the bus.

The message generation by a task executing on a processor is independent of the bus access mechanism used. A task generates a real-time message, waits to get control of the bus, and resumes execution only after transmitting the message. As stated above, the intermessage generation time is assumed to be an exponentially distributed random variable with a mean intermessage interval that is a function of the task's invocation period. Thus, tasks with different invocation periods will have different mean intermessage generation times.

The time and mechanism involved in putting a message on the bus is independent of the bus access mechanisms used. The travel time of a message from one processor another depends on the bit rate of the bus and the physical distance separating these two processors. We shall not analyze this parameter since it is not affected by the bus access mechanism and hence does not make any difference in contrasting the performance of the token bus with that of the polled bus. For similar reasons, the message generation by the task will not be considered any further.

The time required to service a request is independent of the bus access mechanism used. It is assumed to be a random variable, which is identically distributed for all the tasks in the workcell. The service time distribution directly affects the number of message deadlines missed in a cell. Hence, any assumptions about this distribution shall be deferred until all the parameters are analyzed. For the time being it suffices to assume that the service time is identically distributed for all the tasks of a cell.

The parameter of our chief interest is the *bus access time* or the time taken to access the bus, since it directly determines the number of message deadlines missed due to the unavailability of the bus. Let $W_i$ denote the bus access time or wait time for a task $T_i$. We shall derive an expression for the wait time for the message generated by a task.

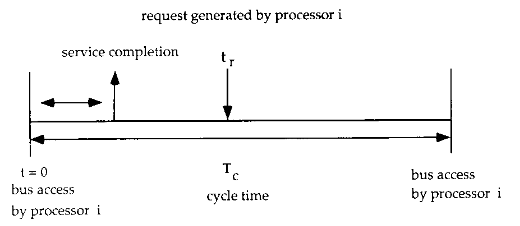


Fig. 8. The cycle time at a processor in the workcell.

The events occurring at a single processor are depicted in Fig. 8. (We are now analyzing the system from the viewpoint of a single processor.) At time $t = 0$, the processor had last successful possession of the bus. The processor may have used the bus or just passed the token along to the next processor in the case of the token bus. The random variable $T_c$ or the *cycle time* is defined as the time interval from a processor's first possession of the bus to its next possession of the bus. The cycle time depends on whether or not the other processors use the bus while the token completely cycles once around all the processors attached to the bus.

A message is generated by task $T_i$ at time $t_i^r$. Let $S_i$ be the time required to service the message, which is the time taken to deliver the message to the destination once the bus has been acquired by the sending task. As stated earlier, the service times are assumed to be identically distributed for all the tasks of a cell. Let $M$ be the number of tasks in the cell under consideration—which is the same as that of processors in the cell—and let $\lambda_i$ be the rate at which messages are generated by $T_i$.

If there are not requests generated by any of the tasks on the token bus, the cycle time—the time taken for the token to reappear at a processor after traversing all the other processors in a predetermined fashion—is given by $T_c = T_{\text{ring}} + T_{\text{token}}$, where $T_{\text{ring}}$ is the time taken for the token to travel on the bus once around the logical ring of processors and $T_{\text{token}}$ is the time taken for a processor to recognize the token and put it back on the bus if it does not have any message to transmit. The maximum value of $T_c$ possible occurs when all the tasks at a cell have messages to send. Thus

$$T_{\text{ring}} + T_{\text{token}} \le T_c \le T_{\text{token}} + T_{\text{ring}} + \sum_{i=1}^{M} S_i.$$

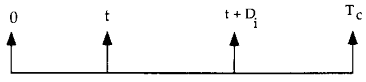
First, we shall determine the cycle time $T_c$ in terms of the service times. To compare the performance of the polled bus with that of the token bus, the cycle time $T_c$ perceived by all the processors is assumed to be the same, since a mean cycle time, as opposed to an instantaneous cycle time at each processor, will be considered for the purpose of comparison. Assuming $T_c < P_i$ where $P_i$ is $T_i$'s invocation period, the probability that $T_i$ will generate a message during $T_c$ is given by $\int_0^{T_c} \lambda_i e^{-\lambda_i t}\, dt$.

The cycle time $T_c$ can then be expressed as

$$T_c = \left( \sum_{i=1}^{M} S_i \int_0^{T_c} \lambda_i e^{-\lambda_i t}\, dt \right) + T_{\text{ring}} + T_{\text{token}}$$
$$= \int_0^{T_c} \left( \sum_{i=0}^{M} S_i \lambda_i e^{-\lambda_i t} \right) dt + T_{\text{ring}} + T_{\text{token}}. \tag{2}$$

The above equation can be simplified to

$$\sum_{i=1}^{M} S_i e^{-\lambda_i T_c} + T_c = \sum_{i=1}^{M} S_i + C \tag{3}$$



t : message generation time.

t + $D_i$ : deadline for the message.

$T_c$ : cycle time for the token bus.

Fig. 9. Events occurring during a token cycle.

where $C$ is some constant. The above expression can be solved for $T_c$ if the service time distributions are known. The probability of a message generated by a task $T_i$ missing its deadline can now be calculated. Let $d_i^r$ be the deadline of $T_i$'s current message generated at time $t_i^r$. Let $D_i$ denote the *relative deadline* ($d_i^r - t_i^r$) of $T_i$'s current message. The events occurring in a cycle are shown in Fig. 9. In this figure, the processor last had access to the bus at $t = 0$. The next access will occur at $t = T_c$, where $T_c$ is the cycle time computed above. A message is generated by task $T_i$ at time $t \le T_c$ and has a relative deadline $D_i$.

$T_i$'s current message will miss its deadline if and only if $t + D_i \le T_c$. Because of the assumed memoryless property of message arrivals, the probability of a message being generated by a task in time $t$ is given by $\lambda_i e^{-\lambda_i t}$. The probability of $T_i$'s current message missing its deadline can be expressed as

$$P_{md}^i = \int_{t=0}^{T_c} \lambda_i e^{-\lambda_i t} P(D_i \le T_c - t)\, dt. \tag{4}$$

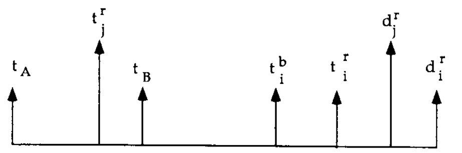
## B. The Polled Bus

In this subsection, we shall analyze the performance of the polled bus, access to which is determined based on the poll numbers generated by the various competing tasks. In particular, the probability of a message missing its deadline will be analyzed. As before, each processor in a cell is assumed to execute only one task during the time interval of interest. This task is characterized by its invocation period and its user-assigned priority, which comprises the second field of the poll number.

In order to facilitate our analysis, we shall use the following notation. (Any notation found in this section that has not been defined here will be the same as the notation used in the previous subsection.)

| | |
|---|---|
| $t_i^r$ | Time at which the current or $r$th message of task $T_i$ was generated. |
| $t_j^r$ | Time at which the current or $r$th message of task $T_j$ was generated. |
| $d_i^r$ | Deadline for the current or $r$th message of task $T_i$. |
| $d_j^r$ | Deadline for the current or $r$th message of task $T_j$. |
| $t_i^b$ | Time at which the bus was last acquired by task $T_i$. |

The most recently generated message of a task whose deadline has not yet expired is termed as the *current message* of the task. After the expiration of the deadline, the message loses its meaning. The various events that are significant in the analysis are depicted in Fig. 10. At $t_i^b$, $T_i$'s current message had the highest poll number or the closest deadline. In the absence of any competing messages from other tasks, $T_i$'s current message generated at $t_i^r$ will be serviced. If there are other competing current messages from other tasks with closer deadlines or larger poll numbers than $T_i$'s current message, then $T_i$'s current message will have to wait for the bus until all the other messages



$$t_A = t_i^r - (d_j^r - t_j^r)$$

$$t_B = d_i^r - (d_j^r - t_j^r)$$

Fig. 10. Event times in the case of a polled bus.

with closer deadlines have been serviced. The time spent by $T_i$ in waiting for the bus is termed as the *bus access time* or *wait time* and is denoted by $W_i$.

We shall derive an expression for the mean wait time experienced by $T_i$. The deadlines for the messages generated by the various tasks are determined by the method outlined in Section III-C. In Fig. 10 we consider $T_j$'s current message, generated at $t_j^r$, which is in competition with $T_i$'s current message for use of the bus. $T_j$'s current message shall be given the bus in preference to $T_i$'s current message if and only if the deadline $d_j^r$ of $T_j$'s current message is closer than the deadline $d_i^r$ of $T_i$'s current message. In other words, $T_i$'s current message will have to wait for $T_j$'s current message if $d_j^r \leq d_i^r$. Since we have assumed the deadline field to be the most significant field, in most cases[5] the condition on the poll numbers is reduced to the requirements on the deadlines.

From Fig. 10, it is clear that any competing messages from other tasks must have been generated during the time interval $[t_A, t_B]$ in order to gain priority over $T_i$'s current message. The probability of the above event is easily determined as the tasks are assumed to generate messages in a memoryless fashion. Therefore, the expression for the wait time $W_i$ for $T_i$'s current message is given by

$$W_i = \sum_{j \text{ s.t. } d_j^r \leq d_i^r} S_j \left( \int_0^{d_i^r - t_i^r} \lambda_j e^{-\lambda_j t} \, dt \right)$$
$$= \sum_{j \text{ s.t. } d_j^r \leq d_i^r} S_j \left(1 - e^{-\lambda_j (d_i^r - t_i^r)}\right). \tag{5}$$

The probability of $T_i$'s current message missing its deadline can now be determined. Let the relative deadline of $T_i$'s current message be $D_i$ where $D_i = d_i^r - t_i^r$. The probability of $T_i$'s current message missing its deadline is given by

$$P_{md}^i = P(D_i \leq W_i). \tag{6}$$

Equation (6) may be used to arrive at the probability of a message missing its deadline. Performances of the token bus and the polled bus were simulated, and the probability of messages missing deadlines using the token bus was compared with that using the polled bus. It was assumed that all tasks had the same message generation rate and identically distributed message service times for the purpose of comparison. The effects of varying the number of tasks, message service times, and task execution times, for both the polled bus and the token bus, are then simulated and plotted in Figs. 11–13. In these simulations the deadline field of the poll numbers were assumed to have the actual deadline, not some quantized version of the deadline. It is expected that the performance numbers will be less optimistic for the finite resolution case. Hence, these simulations represent the maximum possible performance achievable by using the poll-number-based bus access scheme (with the deadline field being the most significant).

[5] This is not true when the deadlines are equal.

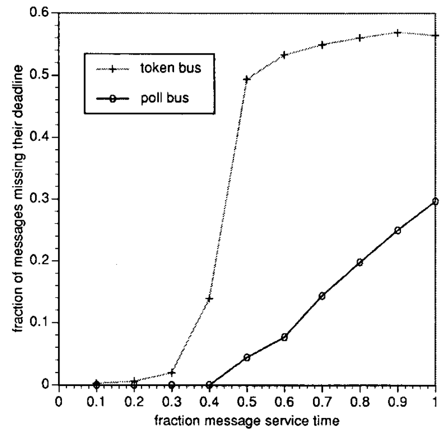


Fig. 11. Effect of message service time on fraction of messages missing deadlines.

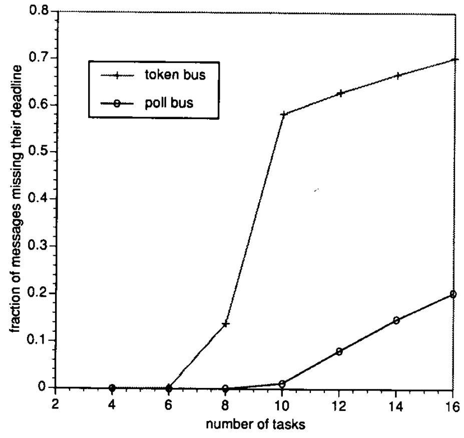


Fig. 12. Effect of number of tasks on fraction of messages missing deadlines.

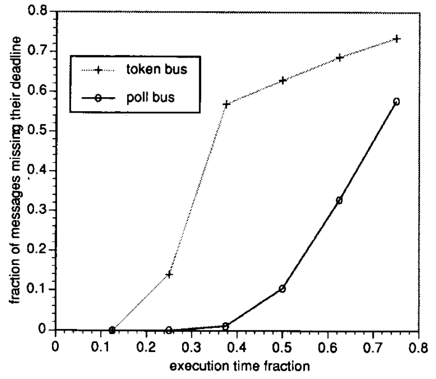


Fig. 13. Effect of execution time of fraction messages missing deadlines.

The fraction of messages missing their deadlines for various message service times—the time taken for the message to travel to the destination on the bus after gaining the access to the bus—is shown in Fig. 11.

The number of tasks was fixed at eight, the tasks all had the same period (40 ms), and the same execution time of 10 ms. The fraction of messages missing their deadlines increased as message service times for both the token and polled buses were increased. As expected, the polled bus had a much lower fraction of mssages missing their deadlines for all message service times than the token bus.

In Fig. 12, the fraction of messages missing deadlines for different cell configurations (number of tasks at the cell) is shown. The fraction of messages missing their deadlines increased with the increase in the number of tasks for both token and polled buses. Again, the polled bus had a much lower fraction of messages missing their deadlines for all cell configurations than the token bus.

The fraction of messages missing their deadlines for different execution times of the tasks is shown in Fig. 13. The message service time and the message generation rate were assumed to be the same for all the tasks and were fixed. There were eight tasks at the cell. As before, all tasks were periodic with a period of 40 ms. The fraction of messages missing their deadlines increased with increase in the execution time of the tasks. For all task execution times, the polled bus resulted in a lower fraction of messages missing their deadlines than the token bus.

## V. Conclusion

In any CIM system, the probability of a control task missing its deadline should be kept as small as possible. CIM tasks often communicate with one another in order to collectively perform some useful functions, such as assembly and/or material handling operations. It is essential that messages sent by various tasks be delivered before their deadlines in order for the tasks to complete their execution in time. We have proposed and analyzed a new bus access mechanism that uses a poll number to minimize the probability of messages missing their deadlines. This mechanism is shown to be significantly better than the token bus protocol in meeting message deadlines. The simplicity, flexibility, decentralization, and the performance improvement offered by the poll number approach make it particularly suitable for the control and coordination of real-time devices in a workcell of CIM systems.

## Acknowledgment

The author is grateful to Y. Muthuswamy for his contribution to the performance evaluation of the polled bus in Section IV and anonymous reviewers for their constructive comments.

## References

[1] A. Ray, Ed., *Proc. NSF Workshop on Computer Networking for Manufacturing Systems* (Pennsylvania State Univ.), Nov. 1987.

[2] A. S. Tanenbaum, *Computer Networks*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[3] "Manufacturing Automation Protocol (MAP) specification (draft)," Feb. 1986.

[4] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. Ass. Comput. Mach.*, vol. 19, no. 7, pp. 395-404, July 1976.

[5] S. L. Beuerman and E. J. Coyle, "The delay characteristics of CSMA/CD networks," *IEEE Trans. Commun.*, vol. 36, no. 5, pp. 553-563, May 1988.

[6] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and its application," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 4, pp. 357-366, Apr. 1987.

[7] R. H. Douglas, "IEEE token bus LAN implementation considerations," in *Proc. IEEE COMPCON*, 1984, pp. 258-260.

[8] D. W. Jacobson, "High performance reliable token bus for the map network architecture," in *Proc. IEEE Conf. Local Comput. Networks*, 1986, pp. 26-33.

[9] W. M. Gentleman, "Message passing between sequential processes: The reply primitive and the administrator concept," *Software Practice Experience*, vol. 11, pp. 435-466, 1981.

[10] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state change broadcasts," *IEEE Trans. Comput.*, vol. 38, no. 8, pp. 1124-1142, Aug. 1989.

[11] J. T. Quatse, "An architecture for real-time cell control," *Contr. Eng.*, vol. 34, no. 5, pp. 56-64, May 1987.

[12] *IEEE Standards for Local Area Networks: Token-Passing Bus Access Method and Physical Layer Specifications*, ANSI/IEEE Standard 802.4-1985, 1985.

[13] J. F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-access protocols and time-constrained communication," *ACM Comput. Surveys*, vol. 16, no. 1, pp. 43-70, Mar. 1984.

[14] G. L. Choudhury and S. S. Rappaport, "Priority access schemes using csma-cd," *IEEE Trans. Commun.*, vol. COM-33, no. 7, pp. 620-626, July 1985.

[15] I. Chlamtac, A. Ganz, and Z. Koren, "Prioritized demand assignment protocols and their evaluation," *IEEE Trans. Commun.*, vol. 36, no. 2, pp. 133-143, Feb. 1988.

[16] T. B. Smith and J. H. Lala, "Development and measurement of fault-tolerant multiprocessor (FTMP)," NASA Rep., vol. I, May 1985.

[17] K. G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious failure," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 2-12, Jan. 1987.

[18] K. G. Shin and M. E. Epstein, "Intertask communications in an integrated multi-robot system," *IEEE J. Robotics Automat.*, vol. RA-3, no. 2, pp. 90-100, Apr. 1987.

**Kang G. Shin** (S'75-M'78-SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, NY. In 1982, he joined the University of Michigan, Ann Arbor, where he is currently a Professor of Electrical Engineering and Computer Science. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called HARTS, to validate various architectures and analytic results in the area of distributed real-time computing. He has held vising positions at the U.S. Airforce Flight Dynamics Laboratory; AT&T Bell Laboratories; the Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley; and the International Computer Science Institute, Berkeley, CA. He has authored or coauthored over 180 technical papers in the areas of fault-tolerant computing, distributed real-time computing, computer architecture, and robotics and automation.

Dr. Shin was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, and the Guest Editor of the 1987 August special issue on real-time systems of IEEE Transactions on Computers. He is currently a Distinguished Visitor of the IEEE Computer Society and an Area Editor of the *International Journal of Time-Critical Computing Systems*. In 1987, he received the IEEE Transactions on Automatic Control Outstanding Paper Award for a paper on robot trajectory planning. In 1989, he received the Research Excellence Award from the University of Michigan.