Dynamic Scheduling for
Soft Real-Time Distributed Object Systems

V. Kalogeraki, P. M. Melliar-Smith and L. E. Moser
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
vana@alpha.ece.ucsb.edu, pmms@ece.ucsb.edu, moser@ece.ucsb.edu

Abstract

Distributed real-time applications require flexible and dynamic scheduling mechanisms to provide timeliness guarantees to application objects. In this paper we present a new scheduling algorithm that exploits the task laxities and the object importances to make effective scheduling decisions. The algorithm uses current timing and resource measurements to determine the feasibility of the tasks and to distribute the objects to the processors. A task’s timing parameter (laxity value) is carried, from one processor to another, with the object invocations, yielding a system-wide scheduling strategy that requires only local computations. The algorithm aims to ensure that a low importance object does not delay the execution of a high importance task.

1 Introduction

Many modern computer applications are built as real-time distributed object systems. The correctness of a real-time system depends on both the logical result and the timeliness of the computations. Hard real-time systems require both functionally correct executions and results that are produced on time. Hard real-time systems are designed to ensure that there are no missed deadlines, often at the expense of resource utilization and average performance. Soft real-time systems allow an occasional violation of a timing requirement, provided that it does not result in system failure. Soft real-time systems aim to meet as many deadlines as possible before attempting to maximize the average performance.

For multi-processor environments, it has been shown [2] that no scheduling algorithm is optimal without a priori knowledge of the deadlines, computation times and arrival times of the tasks. Even simple heuristics [17] that account for resource requirements significantly outperform popular algorithms, such as earliest deadline first scheduling and least laxity scheduling, which ignore resource requirements. Most scheduling algorithms in the literature assume that this knowledge is available in advance and generate preplanned schedules for the tasks. However, complex distributed applications require flexible and dynamic scheduling algorithms that provide timeliness guarantees to the application objects.

In this paper we present a dynamic scheduling algorithm that leverages the flexibility and interoperability provided by the Common Object Request Broker Architecture (CORBA) [13] to support the execution of tasks in a soft real-time distributed object system. The scheduling algorithm examines the computation times, real times and resource requirements of the tasks to determine a feasible schedule for the tasks. All scheduling decisions are made dynamically, and the algorithm tries to guarantee that the timing requirements of the applications are met. The schedule is driven by the urgency of the tasks and the importance of the tasks and by the objects invoked by the tasks. The task’s invocation of an object carries the task’s timing parameters (laxity value) with it from one processor to another, yielding a system-wide scheduling strategy that requires only local computations.

2 System Model

The model of our system is shown in Figure 1. There is a single Global Scheduler for the system, responsible for computing the initial priorities (laxities) for the tasks, based on information provided by the application programmer. The Global Scheduler works in concert with a Resource Manager. As new tasks are introduced into the system, the Global Scheduler uses current system information, collected by the Resource Manager, to distribute the objects on the processors and to make configuration decisions. The Resource Manager maintains current system information to balance the load on the resources, migrating objects when necessary, to maintain a uniform load on the processors.
A Local Scheduler on each processor is responsible for specifying a local ordered list (schedule) for the objects on the processor which defines how access to the resources is granted. A Local Dispatcher selects the most eligible object from the list to be executed next. A Profiler on each processor measures the usage of the resources and monitors the behavior of the application objects and supplies this information to the Resource Manager. The Global Scheduler bases its allocation decisions on this feedback information. This scheduling mechanism has been incorporated into our Resource Management System [8] and is used to schedule CORBA objects.

The Object Management Group (OMG) has adopted the Real-Time CORBA architecture [14] and has issued a Request for Proposals (RFP) for Dynamic Scheduling [15]. Responses to the Dynamic Scheduling RFP are expected to propose interfaces and mechanisms to extend the fixed-priority scheduling policies defined in the Real-Time CORBA architecture to support applications that require dynamic resource scheduling.

### 2.1 Task Metrics

An application task \( t \) consists of a sequence of object invocations. The execution of the task is triggered by an external input (which can be either a timer signal or the completion of another operation) and can result in the generation of an external result as the outcome of the task’s operation. Essentially, to schedule a task we schedule the objects invoked by the task.

The arrival time of aperiodic tasks, which are activated when certain events occur, is not known a priori; tasks are scheduled dynamically as they arrive. This contrasts with static systems where the schedules of tasks are determined in advance and remain fixed while the tasks execute.

Although tasks arrive independently, they are not necessarily disjoint. Thus, although an application object is created by a single application task it is capable of invoking methods on objects of other tasks. Each task \( t \) is associated with the following parameters:

- \( \text{Deadline}_t \): the time interval, starting at task initiation within which task \( t \) should be completed, as specified by the application designer
- \( \text{Importance}_t \): a metric that represents the relative criticality of task \( t \), as specified by the application designer. Under overload conditions, when it is not possible to meet all deadlines, low importance tasks are dropped.
- \( \text{ProjectedLatency}_t \) or \( \text{ComputationTime}_t \): the estimated amount of time required for the task to complete, as measured by the Profilers
- \( \text{Laxity}_t \): the difference between \( \text{Deadline}_t \) and \( \text{ProjectedLatency}_t \), a measure of urgency of task \( t \), calculated by the Global Scheduler. The Local Scheduler schedules the objects of task \( t \) according to \( \text{Laxity}_t \), which it dynamically adjusts as the task executes.

Tasks are also characterized by the resources they require. The scheduling algorithm resolves the contention over the CPU as a resource so that the deadlines of the tasks are met. Other resources, such as memory or disk bandwidth, are not considered.

### 2.2 Object Metrics

Each object consists of a set of methods. The method invocations among the objects are recorded by the Profilers and stored in the application task graph, which describes the relationships among the objects and represents the flow of operations for each task. Each object \( i \) is described by:

- \( \text{ObjectImportance}_i \): the importance of object \( i \) to the application task \( t \) that invokes the object. This metric is used to increase the probability that a high importance object is scheduled before a low importance object.
- \( \text{MeanProcessingTime}_i \): the mean processing time \( \tau_{mp} \) required for each method \( m \) of object \( i \) to execute on processor \( p \), excluding queueing time or the time required for embedded invocations of other methods.

An object with low importance can delay the execution of a high importance task. Object importance is used to
ensure that an object, currently executing a method for a low priority task, does not delay a request from a high priority task that is currently waiting for access to the object. The high importance object will be executed before the low importance object, when their laxity values indicate that both objects will meet their deadlines. Factors such as data locality and communication overheads between the objects, which can affect the distribution of the objects, and, subsequently, the performance of the system, are not considered.

3 The Scheduling Strategy

The challenge in a distributed object system is to guarantee that tasks will meet their deadlines, given object dependencies and resource requirement constraints. In soft real-time systems, the completion time of the tasks is not an absolute constraint. A soft real-time requirement deadline normally implies good response time, but missing some soft deadlines is tolerable.

Least laxity scheduling assigns the higher priority to the tasks with the minimum laxity values. The laxity of a task is defined as the difference between its deadline and computation time. The optimality of least laxity scheduling compared to earliest deadline first (EDF) scheduling, is due to the fact that least laxity scheduling considers the computation times of the tasks to derive the laxity values. However, least laxity scheduling uses only timing measurements and, therefore, fails to distinguish between important and less important tasks and objects. Given that an application task involves multiple objects distributed in different processors, the question is how one can guarantee that a low importance object does not delay the execution of a high importance task causing it to miss its deadline. Even if the laxity value is large enough to guarantee that an object can be scheduled locally before the task’s deadline, it is not sufficient to guarantee that the remaining objects of the task will indeed complete within the task’s deadline.

Timing requirements and importance metrics are not normally correlated. Consequently, scheduling tasks to maximize the probability that the most important tasks meet their deadlines is a non-trivial problem. One approach would be to use the laxity value to schedule the objects on the processors and use the object importance metric to decide which objects could be abandoned in the case of processor overload. Such a scheme fails to guarantee our objective.

Our scheduling algorithm relaxes the timing constraint (laxity value) and considers the object importance when scheduling the tasks on the processors. The algorithm schedules the tasks based on their laxity values values and at the same time ensures that a high importance object will be scheduled before a low importance object.

3.1 Global Scheduler

When a new task arrives, the Global Scheduler computes the initial laxity of the task as:

\[
\text{Laxity} = \text{Deadline} - \text{Projected Latency}
\]

where Deadline is the time within which the task should be completed and Projected Latency is the estimated time to task completion. This information is stored in the task graph, kept by the Resource Manager, along with other information about the application task. As the task executes, the objects of the task are scheduled on the local processor according to the remaining laxity of the task. The laxity value is updated based on the estimated computation time of the objects and the actual time required for their execution. The Local Scheduler subtracts from the remaining laxity Laxity, the difference between the actual time ProcessingTime; measured by the Profilers and the MeanProcessingTime; calculated based on previous executions of object i. The remaining laxity of the task is used to schedule the remaining objects of the task. If the object execution is completed more quickly than was projected, the task laxity increases. If the execution completes more slowly, the task laxity is reduced and the scheduling priority of the task increases.

If the time between the task’s deadline and computation time is short, then the laxity is small. We focus on tasks with relatively small laxity values and multiple importance levels. The Global Scheduler estimates the Queueing Latency [8] of all the objects on the processors based on the method invocations and the load measurements reported by the Profilers. In the absence of more specific information about the task behavior, we assume that the processing time of the methods follows an M/M/1 queueing model.

3.1.1 Object Distribution

A task’s computation time can be influenced by data dependencies and hardware load fluctuations and may therefore vary from execution to execution. To make accurate decisions, the Global Scheduler needs preplanned projections of the objects on the processors. In a dynamic real-time system where detailed or complete object information may not be available, mechanisms are required to allow the scheduling of objects on a millisecond basis. Given that the Global Scheduler tries to preserve the execution of the tasks currently scheduled, the Global Scheduler determines whether the new objects can be scheduled by computing the effects of the increased load on the latencies of the existing tasks.

Our algorithm yields a system-wide strategy that requires relatively little computation. The Global Scheduler uses the task graph to define the objects to be distributed, starting from the object consuming the highest processing load. Then, it computes the contributions to the queueing length...
of the least loaded processor as the result of the addition of the new object. Essentially, the new object will increase the load on the processor and will increase the queueing latencies of all objects located on the processor. The new object can be accepted only if the queueing length of the processor remains in a reasonable range, compared to the queueing lengths of the other processors.

### 3.2 Local Scheduler

The Local Scheduler accepts requests for objects to be executed locally on the processor. Upon receiving such a request, the Scheduler uses the laxity value of the object to determine whether the object can be scheduled on the processor. Whether or not the object will meet its deadline, depends on the computation times of the objects currently in the ready queue.

To determine whether the admission of the new object will affect the laxities of the objects currently on the processor the Local Scheduler uses the following condition:

\[
\sum_{j=0}^{i-1} CT_j \leq L_i
\]

where \(CT_j\) is the estimated computation time for object \(j\). Table 1 illustrates how the Local Scheduler uses the objects’ laxity values to order the objects in the ready queue. Each object has a laxity value greater than the sum of the computation times of the preceding objects.

While the above condition guarantees that the new object will be scheduled on the processor, this condition is not sufficient to guarantee that the task will meet its deadline. The laxity value of the object represents the urgency of the task that invokes the object. Our goal is to increase the probability that high important tasks meet their deadlines. Therefore, the order in which the object will be dispatched is driven by the object’s importance.

### 3.3 Scheduling Function

The Local Scheduler uses a Scheduling function to determine the real-time priority of the object on the processor. The Scheduling function positions the object in the ready queue based on both the laxity value and the importance of the object. This function guarantees that a high-priority object will be scheduled before a low-priority object when the laxity values indicate that both objects will meet their deadlines. The scheduling function (Figure 2) consists of two phases: the object placement phase and the real-time priority assignment phase.

#### 3.3.1 Ready Queue

The Local Scheduler maintains a ready queue with the objects to be scheduled on the processor. The ready queue can be viewed as an array of lists where each element of the array represents a range of laxity values and where all of the objects with similar laxity values are linked together. The lists are ordered according to non-increasing laxity values. The first element in the list is the object with the highest importance.

In the object placement phase, the Scheduler uses the formula from Section 3.2 to determine an initial position of the new object in the ready queue. The Local Scheduler places the object in the array list that corresponds to the object’s laxity value. If no such list exists, the Local Scheduler creates the list and places the object at the head of the list.

Once the new object is inserted in an initial position according to its laxity value, the Local Scheduler, using the object’s importance, tries to move the object to a higher position in the ready queue. To do that, the Scheduler compares the priority of the new object with the priority of the object at the previous position in the ready queue. If the new object has a higher priority value, then the Scheduler attempts to swap the execution order of the two objects. This can be achieved only if the laxity value of the object

### Table 1: Computation times, laxity values and importance values. The objects are ordered according to the laxity values of the tasks that invoke those objects.

<table>
<thead>
<tr>
<th>Obj(i)</th>
<th>CT(i)</th>
<th>(\sum CT_j)</th>
<th>Lax(i)</th>
<th>Imp(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>13</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>20</td>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>22</td>
<td>50</td>
<td>2</td>
</tr>
</tbody>
</table>
in the previous position in the queue can accommodate the computation time of the newly inserted object.

Now, assume that the new object has been inserted in the ready queue position $i$ according to its laxity value. The previous position in the ready queue is occupied by object $i-1$. Object $i-1$ has been inserted in the queue based on its laxity value. Our Scheduling algorithm guarantees that the laxity value of object $i-1$ is greater than the sum of the computation times of objects 1 through $i-2$. To be able to swap objects $i$ and $i-1$, we must ensure that the laxity value of object $i-1$ can also accommodate object $i$. This is guaranteed only if

$$\sum_{j=0}^{i-2} CT_j + CT_i \leq L_{i-1}$$

If this condition holds, the Local Scheduler can swap the two objects in the ready queue. The Scheduler continues the same procedure until the new object cannot move to a higher position in the ready queue, either because the next object in the ready queue has higher importance or because its laxity is too low to allow the interchange.

Table 2 shows the final order of the objects of Table 1 in the ready queue. The Scheduling function has determined the new order based on both the laxity value and the importance of the objects.

### 3.3.2 Real Time Priority

Once the Local Scheduler determines the final position for the new object in the ready queue, it computes the real-time priority of the object. In priority-based preemptive scheduling, at each moment, the object that executes is the one with the highest priority. Intuitively, the highest priority is assigned to the object with the least laxity value and the highest importance. If an object with low priority executes, it can be preempted by objects with higher priorities.

Assuming that object $i$ with object importance $imp_i$ is invoked by task $t$ with laxity value $laxity_t$, the Local Scheduler computes the real-time priority $rt\_priority_i$ of the object $i$ as:

$$rt\_priority_i = \frac{c}{laxity_t} + \frac{1}{imp_i}$$

where $c$ is a constant used to weight the laxity and the importance. The reason is that the laxity can take a wide range of values (typically from 1 to 400 ms), while importance takes only a small number of values (from 1 to 5). The choice of the value of $c$ depends on the relative significance of laxity and importance to the application.

The higher priority is assigned to the objects with the larger $rt\_priority$ values. These are the objects with the maximum urgency (smaller laxity value) and higher importance (smaller absolute importance value). Using the above formula, with $c = 5$, an object $A$ with $laxity_A = 8$ and $imp_A = 1$ is assigned a higher $rt\_priority_A = 1.27$ than an object $B$ with $laxity_B = 5$ and $imp_B = 5$ ($rt\_priority_B = 1.09$).

The real-time priorities of the objects are stored in the ready queue along with the laxity values of the objects. As the priority values increase, more objects are assigned to those real-time priority values. Typical priority scheduling mechanisms have a fixed number of levels. Therefore, tasks with different laxity values are mapped into the same level.

### 3.4 Local Dispatcher

The Local Scheduler works in concert (Figure 3) with the Local Dispatcher. The Local Dispatcher is responsible for granting access to the most eligible object on the processor. This is the object located at the front of the ready queue. At any moment, the object that is currently running in the CPU is the one with the minimum laxity value and the highest importance among the objects on the processor.

The real-time priority defined for the object is used to set the priority of the thread in which it is dispatched. Realtime CORBA defines the Threadpool abstraction to manage threads of execution. Threads can be defined and controlled in a uniform manner, independently of the operating system and platform. A pool of threads can be used to buffer the object requests. Each thread pool is associated with a RTCORBA::Priority value. The RTCORBA::Priority value specifies a static priority for the threads. To accommodate dynamic scheduling schemes, dynamic priorities must be associated with the threads so that a thread can assume the appropriate priorities for the task that it is executing.

A preemptive algorithm may have a high cost due to the high schedulability it provides. If the object currently executing is close to completion, the Local Dispatcher may decide to allow the currently running object to finish execution. This minimizes context switching overheads.

The Dispatcher triggers the execution of an object if the real-time priority of that object is higher than the priority of the object currently executing, preempting the execution.
of the lower priority object. If the current object finishes execution, the Dispatcher starts the next highest priority object.

Our algorithm takes advantage of the scheduling functions the operating system provides. Our Local Dispatchers are responsible for setting the priority of the threads that execute the objects. These priorities are mapped into specific real-time priorities supported by the operating system. In a general-purpose operating system, such as Solaris, we do not have control over kernel-internal activities.

The Solaris operating system supports three scheduling classes (in the range of 0-159) that define policies and algorithms applied to the processes and threads belonging to the particular scheduling class. These classes are: the timeshare (TS) scheduling class designed to provide fairly even distribution of the hardware processor resources to the processes and threads, the interactive (IA) scheduling class designed to provide snappier behavior for desktop systems, and the real-time (RT) scheduling class that provides some degree of real-time application support. Threads in the real-time scheduling class have a higher priority (in the range of 100-159) than threads belonging to the other classes and they run until they voluntarily surrender the processor.

Our Dispatcher assigns priorities from the real-time scheduling class provided by the Solaris operating system. We use the \texttt{priocntl} system call to manipulate the scheduling priorities. The advantage is that these are deterministic priorities and that they will run to completion or will be interrupted only by higher real-time priority threads. This has the effect of bounding the priority inversions. Although Solaris provides an environment for many real-time applications, it is a general-purpose operating system. An operating system (such as POSIX) designed for real-time applications would provide more appropriate scheduling interfaces, along with other features, for supporting real-time applications.

### 3.5 Residual Laxity

A task with negative laxity value indicates that the task has missed its deadline. In a soft real-time system the task continues to execute even if its deadline is missed. Our strategy adjusts, however, the order in which such tasks are scheduled.

Our primary goal is to guarantee that tasks meet their deadlines. Thus, we schedule the objects of a delayed task, only if the computation times of those objects indicate that their timing margins are sufficient to accommodate the delayed task.

When a task completes, the remaining laxity of the task is recorded as the \textit{Residual Laxity} of the task and is compared with the initial laxity of the task, as computed by the Global Scheduler before the task starts execution. The residual laxity of the task should be the same as the task’s initial laxity, indicating that the projected latency, as measured by the Profilers during the previous executions of the task, was a good estimate of the computation time of the task. The residual laxity of the task is stored by the Resource Manager in the task graph and the Global Scheduler uses the ratio of the residual laxity to the initial laxity to adjust the estimates of the projected latency.

### 4 Run-Time Monitoring

The Profilers in our system are specific to individual processors and measure the current utilizations of the processor’s resources and, therefore, can detect significant deviations in performance.

The Profilers monitor the method invocations to build profiles for the objects on the processors. When an invoked method completes and returns its response to the invoking local method, the Profiler on the processor notes the time and calculates an actual time for the invocation from the time at which the method was invoked and the time at which the response was received. This actual time is subtracted from the projected time and the difference is added to the remaining laxity of the task.

When the local object finishes execution, the computation time is stored in the object’s profile and the estimated computation time is updated. The Profiler builds profiles for the objects based on the history of their executions.
The Profiler uses exponentially weighted averaging to determine the mean values of the measurements. The mean value of each measurement is determined by a sequence of measurements with greater weight being given to more recent measurements. If the system is stable and the number of invocations is large, the above calculation gives a good approximation to the mean value. If the behavior changes dynamically, a large weight for the latest measurement yields rapid response to changing conditions.

An object may require the use of multiple types of resources at different time intervals throughout its execution. Each Profiler measures the current utilization of the processors resources (i.e., CPU, memory, disk etc) and, periodically, provides feedback information to the Resource Manager. The Global Scheduler uses the current load on each processor to estimate the Queueing latency for the objects on the processors. In addition, the Global Scheduler uses the resource measurements to make efficient allocation decisions. In the case of overloaded resources or lack of resources due to failures, the Resource Manager may decide to migrate objects to different processors to free computing resources [8].

5 Related Work

Research into scheduling has been dominated by hard real-time systems. Most of the scheduling work reported in the literature have been based on static scheduling algorithms.

Useful results also exist for soft real-time distributed systems. Jensen et al [7] propose soft real-time scheduling algorithms based on application benefit, obtained by scheduling the applications at various times with respect to their deadlines. Their goal is to schedule the applications so as to maximize the overall system benefit. Stankovic et al [18] discuss the Spring Kernel developed for large complex real-time systems. They classify the tasks based on their importance and timing requirements and use value-based functions to drive the schedule. Lawrence [9] proposes a unified quantitative information model that defines information as a function of timeliness, precision and accuracy. He uses this QoS based scheduling scheme to achieve greater efficiency and effectiveness in resource management.

Dertouzos and Mok [2] and Hong et al [6] have shown that least-laxity scheduling is an effective strategy for real-time distributed systems. Gupta et al [4] have investigated scheduling algorithms based on compact task graphs. Harbour et al [5] discuss the difficulty of scheduling when tasks consist of a number of subtasks, each executing at a different priority level.

Manimaran et al [10] have shown that efficient dynamic scheduling algorithms require knowledge of real-time task information including the task’s deadline, resource requirements and worst-case computation time. Brandt et al [1] use a QoS based resource manager, called the DQM, to mediate application resource usage so as to ensure that applications obtain the resources that they need to provide adequate performance. DQM uses execution-level information and current system state to determine appropriate QoS allocations for the running applications. Rajkumar et al [16] present a brute-force scheme to find the optimal resource allocation that satisfies multiple QoS dimensions. Nett et al [12] employ a system-wide monitoring service to provide scheduling components with the necessary information. Our system uses run-time monitoring to estimate the projected task latencies and to adjust the allocations based on actual timing and resource measurements.

Several efforts have focused on extending CORBA with real-time capabilities. Wolfe et al [19] support timed distributed method invocations to enable the expression and enforcement of timing constraints in CORBA client/server interactions. Zinky et al [20] have developed an architecture, called Quality of Service for CORBA Objects (QuO). QuO specifies an application’s expected usage pattern and QoS requirements for each object connection, by extending the Interface Definition Language (IDL) of CORBA with a Quality of Service Description Language (QDL). Gill et al [3] have designed a CORBA scheduling service integrated in their high-performance real-time ORB TAO. They have focused mainly on hard real-time systems using rate monotonic scheduling, but recently have moved to maximum urgency first scheduling, which provides scheduling assurance for critical tasks while offering the flexibility to optimize the use of scarce resources.

Nahrstedt et al [11] have employed resource management mechanisms to provide end-to-end QoS guarantees for multimedia computing and communication. They present a soft real-time scheduler for the Unix environment and a resource broker that provides QoS, negotiation, admission and reservation capabilities for sharing resources, such as memory and CPU. Their dynamic scheduler is based on a preliminary round of testing to capture the behavior of the tasks before the actual execution starts.

6 Conclusion

We have presented a dynamic scheduling algorithm to support the execution of tasks in a soft real-time distributed object system. The schedule is driven by the urgency of the tasks and the importance of the objects to the system. Our algorithm leverages the flexibility and interoperability provided by CORBA, and uses timing and resource measurements to provide timeliness guarantees to the application objects. The scheduling function ensures that a high importance object is scheduled before a low importance object, if their laxity values indicate that both objects will be scheduled within their deadlines.
7 Acknowledgments

This research has been supported by DARPA and AFOSR, Contracts N00174-95-K-0083 and F30602-97-1-0248, and by Rockwell Science Center through the State of California MICRO Program, Grant 96-052.

References


