Dynamic Scheduling of Distributed Method Invocations

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 method invocations require dynamic scheduling algorithms and efficient resource projections to provide timeliness guarantees to application objects. In this paper we present a dynamic scheduling algorithm that examines the computation times, real times and resource requirements of the application tasks to determine a feasible schedule for the method invocations. The schedule is driven by the laxities of the tasks and the importance that the tasks have to the system. Tasks span processor boundaries, and request messages carry scheduling parameters (laxity values) from one processor to another, yielding a system-wide scheduling algorithm that requires only local computations. Experimental results validate our scheduling algorithm, and show that it has minimal overhead.

1. Introduction

Distributed method invocations require flexible and dynamic scheduling algorithms that can provide timeliness guarantees to the application objects. Dynamic scheduling is more applicable to soft real-time systems, where missing a deadline is not catastrophic to the system. In soft real-time systems, the execution times of the methods can vary; therefore, it is difficult to estimate in advance an effective schedule. Scheduling distributed methods becomes more complicated when dynamically invoked methods interact with each another and compete for limited computing resources. Dynamic scheduling algorithms require accurate resource usage projections to ensure timing constraints and achieve the best utilization of the resources. Worst-case allocations are usually not effective, because they trade resource utilization for accurate predictions and can result in underutilized processors.

* This research has been supported by DARPA, AFOSR and ONR Contracts N00174-95-K-0083, F30602-97-1-0284 and N66001-00-1-8931.

In this paper we present a dynamic scheduling algorithm that examines the computation times, real times and resource requirements of the tasks to determine a feasible schedule for the method invocations on the processors. Tasks are modeled as sequences of method invocations of objects located on multiple processors. Our scheduling algorithm is based on the least-laxity scheduling algorithm, which has proven to be an effective algorithm for multiprocessor environments [5]. In least laxity scheduling, the laxity of a task represents a measure of urgency for the task and is computed as the difference between the deadline and the estimated remaining computation time of the task. The schedule is driven by the laxity values of the tasks invoking methods on the objects and by the importance that the tasks have for the system. The decisions are made on a system-wide basis because the tasks consist of methods distributed over many processors. Our algorithm allows a request message containing a method to carry the scheduling parameters of the task from one processor to another, yielding a system-wide scheduling algorithm that requires only local computations. The scheduling algorithm is implemented in the Realize Resource Management system [11, 13] and is used to schedule CORBA [16] objects. Experimental results show that the scheduling mechanism is efficient, and has minimal overhead.

2. Scheduling Model

2.1. Tasks

A task is a sequence of method invocations of objects distributed across multiple processors. A task is executed by a single thread or multiple threads executing in sequence or in parallel on one or more processors. The execution of a task is triggered by a client thread and completes when the client thread finishes execution. Multiple tasks originating from different client threads can be executed concurrently. Similar to our use of the term task, the OMG's Dynamic Scheduling proposal [18] introduces the notion of a dian-
tributable thread as an end-to-end schedulable entity. A distributable thread replaces the concept of an activity that was used in the Real-Time CORBA specification [17].

Tasks enable end-to-end scheduling in that they span processor boundaries and carry scheduling parameters from one processor to another yielding system-wide scheduling strategies that require only local computations. A task's scheduling parameters apply to local threads and methods invoked by the task. The scheduling parameters depend on the scheduling algorithm implemented and can be updated during the execution of the task. For example, the earliest deadline first (EDF) scheduling algorithm uses the deadline of the task as the scheduling parameter throughout the execution of the task, while the least laxity scheduling algorithm (LLS) updates the laxity value as the task executes. The task's resource requirements depend on the resource requirements of the objects invoked by the task, the current state of the objects, and the sequence of method invocations.

2.2. Objects

The object is the smallest scheduling entity in the system. Tasks invoke methods on objects and objects are scheduled based on the scheduling parameters propagated with the methods. Although tasks are triggered independently, they are not necessarily disjoint. Thus, different tasks can concurrently invoke methods on the same objects with different or the same scheduling parameters.

The method invocations of the objects are recorded and stored in the task flow graph (Figure 1) that describes the relationships among the objects and represents the flow of operation for each task. The graph also includes timing and resource information about the individual methods and the tasks as a whole, which are used to guide subsequent executions of the tasks. The computation times of the tasks depend on (1) the previous executions of the tasks, (2) the other tasks running in the system, and (3) the dependencies among them.

3. Scheduling Metrics

3.1. Task Metrics

Tasks are aperiodic and their arrival time is not known a priori. These contrast with static systems where the schedules of the invoked objects are usually determined in advance and remain fixed while the tasks execute. Each task $t$ is associated with the following parameters:

- Task identity: each task has its own identity. A task starts when a client thread is invoked and finishes when the client thread completes execution. Multiple client threads can be executing concurrently, each of them characterized by a different id.
- Task state: describes the current state of the task, that is, whether the task is idle, running or waiting for a resource. Typical task states are: RUNNING, WAITING and INACTIVE.
- Deadline: the time interval, starting at the time the client starts the task, within which the task should be completed, specified by the application developer.
- Importance: a metric that represents the relative importance of the task $t$, specified by the application developer. The importance metric ensures that under overload conditions, when it is not possible to meet all deadlines, low importance tasks either are dropped or are not admitted for execution.
- Projected latency or Computation time: the estimated amount of time required for the task to complete. This time is measured by run-time monitoring functions and includes queuing time. Thus, the Projected latency depends on all the running tasks in the system and the processors where they execute.
- Laxity: the difference between Deadline and Projected latency, a measure of urgency of task $t$. The objects invoked by the task $t$ are scheduled according to Laxity, which is dynamically adjusted as the task executes.

3.2. Object Metrics

Each object is described by a set of methods and the amount of resources required for each method execution. With each object, we associate the following parameters:

- Processor address: the network address of the processor where the object $i$ is located
- Methods: the set of methods of object $i$
- Resource utilization: the percentage of the processing load required for executing methods of the object.
3.3. Method Metrics

For each method $m$ of an object $i$, we maintain:

- **Object$_m$**: the object of which $m$ is a method
- **Method$_m$**: the name of the method
- **Mean$_m$**: the mean time required, after receipt by a processor of a message invoking method $m$, for the method to execute locally on the processor. The Mean$_m$ includes queueing time but excludes the time of invocations of other methods.
- **Mean$_mn$**: the mean time required for method $m$ to invoke method $n$ remotely, including communication and queueing time and also the time of embedded invocations of other methods.
- **Mean$_mn$**: the mean time to communicate an invocation from method $m$ to method $n$ and to communicate the response back. The Mean$_mn$ is computed as the difference between Mean$_mn$ and Mean$_n$.
- **Mean$_mp$**: the mean time required for method $m$ to execute on processor $p$, excluding queueing time and the time required for embedded invocations of other methods.
- **Mean$_mn$**: the mean number of invocations that a single invocation of method $m$ makes on method $n$.

4. Scheduling Architecture

The architecture of our system is shown in Figure 2. There is a single Global Scheduler for the system, which is responsible for computing the initial scheduling parameter (laxity value) for each of the tasks. A task's initial laxity is computed based on information collected during previous executions of the task. If no such information is available, the application programmer must provide an estimated computation time required for the task to run. The initial laxity value is updated during the task's execution, and the objects invoked by the task are scheduled based on this laxity value.

The Global Scheduler is part of the Resource Manager. The Resource Manager is implemented as a collection of CORBA objects, which are distributed and possibly replicated to increase reliability; logically, however, there is only a single copy of the Resource Manager in the system. The Resource Manager maintains current system information, in terms of the location of the objects on the processors and the usage of the resources. As new tasks are introduced, the Global Scheduler uses current system information, collected by the Resource Manager, to distribute the objects on the processors and to make configuration decisions.

A Local Scheduler on each processor is responsible for specifying a local ordered list (schedule) for the method invocations, which defines how access to the CPU resource is granted. The schedule is based on the laxity values of the tasks invoking methods on the objects; the one with the smallest laxity value is placed first in the Scheduler's queue. The laxity value can guarantee whether the method execution will be completed within the task's deadline only if the laxity value of the method invocation on the object is greater than the sum of the computation times of all the method invocations with smaller laxity values currently scheduled for execution on the processor. Note that, because a task consists of method invocations across multiple processors, this condition is not sufficient to guarantee that the task will indeed meet its deadline [12].

A Local Dispatcher selects the most eligible object to be executed next. This is the object whose method invocation has the minimum laxity value, located at the head of the Scheduler's queue. The Dispatcher is then responsible to set the priority of the thread that will carry the execution. This priority will be mapped into a specific real-time
priority supported by the operating system. In an effort to minimize context switching overheads, the Dispatcher may decide to allow the currently executing object to complete, for example, if it is close to completion or if the laxity value of the new object is sufficient to accommodate the execution of the current object.

A Profiler on each processor measures the usage of the resources and monitors the behavior of the application objects, in terms of the execution times of the methods invoked and the resource requirements of the objects, and supplies this information to the Resource Manager. The Global Scheduler bases its allocation decisions on this feedback information.

5. Implementation

5.1. CORBA messages

CORBA [16] provides interoperability between objects located on different processors and implemented in different languages. It uses the General Inter-ORB Protocol (GIOP) and its TCP/IP implementation, called the Internet Inter-ORB Protocol (IIOP). The GIOP protocol supports eight message types used for the communication between the objects, among which are the Request and Reply messages which are most frequently used. A Request message is used to invoke a method on another object. A Reply message is used to respond to a previous request, carrying along the return value of the invocation. A LocateRequest message is used to obtain the current addressing information of the remote object and a LocateReply message is returned in response to the LocateRequest message. Real-Time CORBA includes these two messages as part of the initialization process which eliminates the setup time overhead.

Each method invocation or response, as monitored by our Profilers, is characterized by the following tuple:

\[
<\text{Action, local\_object, invoking\_method, remote\_object, invoked\_method, time\_of\_invocation}>.
\]

where \text{Action} is determined by the Profilers and is one of the following: LOCAL\_START, LOCAL\_COMPLETE, REMOTE\_START, and REMOTE\_COMPLETE. The Profilers distinguish between a remote method invoking a local method on a local object (LOCAL\_START, LOCAL\_COMPLETE) and a local method invoking a remote method (REMOTE\_START, REMOTE\_COMPLETE) on a remote object, and also between the corresponding responses. Our Profilers attach a timestamp to each of the method invocations and can therefore measure the execution and computation times of the local and remote methods as invoked by the tasks.

When a Request message crosses a processor boundary, it carries with it the caller's scheduling parameters (laxity value) in the message header. When a request completes, the scheduling parameters are propagated back to the caller. When the caller receives the Reply message, the actual time required is compared with the projected time and the difference is used to adjust the task's laxity value.

5.2. Scheduling Method Invocations

The Scheduler does not have complete or detailed information of the object methods, such as the code segments (units of code) or the execution loops of the code to make accurate scheduling decisions. To build a Scheduler with such accurate information, off-line schedulability analysis of the objects and their method invocations is required, as is specific code instrumentation. Each time a new task is introduced in the system or completes execution, a new schedulability analysis is performed.

Such an approach is very costly and restrictive. In complex, dynamic distributed object systems, the structure of the system changes dynamically; new tasks are created, existing objects are deleted, and connections between objects are established and completed. Furthermore, the interactions between the objects may vary for different invocations of the tasks. Our Local Scheduler makes scheduling decisions as follows:

- **Local Method Invocation**: The Scheduler decides in favor of a new task invoking a method locally on the processor, only if the new task is more urgent (smaller laxity value) than the currently executing task and the new task's laxity value is smaller than the remaining computation time of the currently executing task. In such a case, the current task will be suspended and the new task will be dispatched for execution. In any other case, the Local Scheduler will order the task in its local queue based on the task's scheduling parameter (laxity value).

- **Local Method Completion**: When a local method finishes execution, the Dispatcher selects the most eligible object to be dispatched next. This is the object whose method execution has the next minimum laxity value. Both the Scheduler and the Dispatcher remain idle until an object is inserted in the queue, \textit{i.e.}, until a task invokes a method locally on the processor.

- **Remote Method Invocation**: When the currently executing method makes a remote method invocation on an object located on the same or a different processor, the CPU should be released for other competing objects on the processor. In such a case, the task's execution propagates to the remote object, and the currently executing method is placed in the Scheduler's queue and ordered according to its laxity value. The object resumes execution only after the method receives the remote response.
• **Deadline Miss:** A negative laxity value indicates that the task will miss its deadline even if the deadline has not passed yet. A soft real-time system allows the task to continue execution even after its deadline, as long as it does not interfere with the execution of the other objects in the processor. Our Local Scheduler checks if there are other objects competing for the processor’s CPU. If not, the currently executing method is allowed to complete; otherwise, it is interrupted and is inserted into the local queue for execution when no other methods with positive laxity values are invoked on the processor.

### 5.3. Method Profiles

The objective of profiling is to measure the actual computation and execution times of the methods and to measure the object’s resource usage during execution. A Profiler on each processor measures the local computation and execution times of the methods on the processor and the communication times to make a method invocation to an object located on the same or a different processor. Our Profilers monitor only method invocations of CORBA objects.

Figure 3 illustrates the method invocation measurements during the execution of the tasks. The Profilers monitor the local execution time for each method, i.e., the time required, after receipt by the Scheduler of a message invoking a method on that processor, to complete the invocation. This includes the time the method spends in the Scheduler’s queue waiting for the CPU to be released, as well as the processing time ($\tau_{mp}$) required for the method to execute locally on the processor. In the absence of more specific information about the task’s behavior, we assume an M/M/1 queueing model for the execution time. Given the load $\rho_p$ on processor $p$, the mean processing time for method $m$ on processor $p$ is given by:

$$\tau_{mp} \equiv (1 - \rho_p) \times \text{Mean Execution Time}_{mp}$$

Similarly, the Profiler estimates the remote method invocations, including communication time, queueing time and also the time for embedded invocations of other methods. We let $\sigma_{mc}$ be the mean transmission time on communication link $c$ for invoking method $m$, excluding queueing delays, and $\rho_c$ be the load on communication link $c$. The mean transmission time for the invocations and responses of method $m$ on communication link $c$ is then given by:

$$\sigma_{mc} \equiv (1 - \rho_c) \times \text{Mean Communication Time}_{mc}$$

From a traversal of the Method Invocation Graph for task $t$ and a classical equilibrium flow analysis, the Resource Manager determines the mean number $x_{tm}$ of invocations of method $m$ for one execution of task $t$. Given the mean processing time and the mean transmission time for method $m$ on processor $p$ and the mean number $x_{tm}$ of invocations of method $m$ made by task $t$, the Resource Manager computes the projected latency for the entire task as:

$$\text{Projected Latency}_t \equiv \sum_{m:p \in mp} \left\{ \frac{x_{tm} \tau_{mp}}{1 - \rho_p} \right\} + \frac{x_{tm} \sigma_{mc}}{1 - \rho_c}$$

where $m \in p$ denotes that the object $i$ of which $m$ is a method executes on processor $p$.

The above information is recorded on a per method basis where the Profilers use their actual measurements as input to the formulas. The timing measurements depend on the parameters of the invocations, the current state of the invoked and invoking objects and the load of the processors on which the objects of the tasks are located.

### 5.4. Queueing Latency

The Global Scheduler estimates the Queueing Latency of all the objects on the processors based on the method invocations and the load measurements reported by the Profilers.

Let $\rho_p$ be the current load on processor $p$. Assuming that $x_{tm}$ is the mean number of invocations of method $m$ of object $i$ for one execution of task $t$ and $\tau_{mp}$ is the mean processing time of method $m$, the Global Scheduler estimates the Queueing Latency for each object as:

$$\text{Queueing Latency}_{tip} \equiv \sum_{m:i \in mc} \frac{x_{tm} \tau_{mp}}{(1 - \rho_p)} - x_{tm} \tau_{mp}$$

$$= \sum_{m:i \in mc} \rho_p \frac{x_{tm} \tau_{mp}}{(1 - \rho_p)}$$

where $m \in i$ indicates that $m$ is a method of object $i$.

The Global Scheduler identifies the objects that produce a high increase in the latency to the completion of a task, by computing the Queueing Ratio of the objects for the tasks. The Queueing Ratio represents the delay in the execution of the object on that processor, because of queueing, and is defined as:

$$\text{Queueing Ratio}_{tip} = \frac{\text{Queueing Latency}_{tip}}{\text{Laxity}_t}$$

Given the queueing ratio of the objects on the processors, the Global Scheduler identifies an overloaded processor as the processor on which the allocated objects, on average, experience a large delay in the latency to the task completion. This is the processor $p$ with a high Queueing Length, defined as:

$$\text{Queueing Length}_p = \frac{\sum_{i:p \in mc} \text{Queueing Ratio}_{tip}}{\text{Number of Objects}}$$
The Queueing Length $p$ is computed by averaging the Queueing Ratio $ip$ of the objects currently on processor $p$. The Global Scheduler uses the Queueing Length $p$ for all of the processors to determine if a particular processor has a queueing length value which is significantly larger than the values of the other processors. Essentially, this would be an overloaded processor, inappropriate for the allocation of new objects and a candidate for the migration of its objects to other processors. Similarly, a small value of Queueing Length $p$ indicates an underloaded processor. Such a processor has a high probability of being able to guarantee the performance requirements of a new object.

5.5. Residual Laxity

When a task completes, the remaining laxity of the task is recorded as the Residual Laxity. The Residual Laxity is then compared with the Initial Laxity of the task, as was measured by the Profilers during the task’s previous executions. Ideally, these two values should be the same, which would indicate that our estimated computation time of the task is accurate.

The Global Scheduler uses the ratio of the residual laxity to the initial laxity to adjust the estimates of the projected latency for the task. For example, if the residual laxity is frequently larger than the initial laxity, this is a strong indication that the processors are overloaded and further allocations should be carefully controlled. Upon completion of the task, the task’s computation time is updated after averaging the previous estimate with the latest measurement.

6. Experiments

Ideally, all of the tasks will have large laxity values and will be able to meet their deadlines. We are interested in investigating the behavior of tasks with small laxity values, when it is easy to miss the deadline in the case of overload.

We have run experiments using the realtime scheduling class of the Solaris operating system. This gives us some element of deterministic control of the applications, because they run to completion unless they are interrupted by higher real-time priority threads. The disadvantage is that, because those threads have a higher priority than all other threads in the system, they can monopolize the CPU and I/O, not allowing any other threads to run. To avoid such situations, we use timer mechanisms and signals to force the real-time threads to surrender the CPU periodically, in order to allow interactive I/O operations to complete. The Scheduler runs in the highest priority of the system and uses the prionct1 system call to manipulate the scheduling priorities.
6.1. Profiler and Scheduler Overheads

The first set of experiments is focused on evaluating the overhead of our local scheduling components. Each local Profiler monitors the behavior of the objects on the processor, in terms of the CORBA Request and Reply messages exchanged among the objects, and measures the percentage of the CPU required for their executions. The Profiler uses TCP/IP sockets to send the feedback information to the Resource Manager. The Global Scheduler then uses this information to make intelligent configuration management decisions.

The time interval between consecutive Profiler measurements depends on the tasks in the system. Previous measurements [11] indicate that careful selection of the monitoring interval is required, because the load fluctuations reported by the Profilers affect the allocation of the new objects on the processors. If the reporting frequency is low, the Resource Manager might not determine any fluctuations. If the frequency is too high, the Resource Manager might detect an increase in the load and proceed with an object migration even though the increase is only transient. Our measurements indicate that a time interval of one second is adequate to capture the load fluctuations in the processor.

The time interval between consecutive Profiler measurements depends on the tasks in the system. Previous measurements [11] indicate that careful selection of the monitoring interval is required, because the load fluctuations reported by the Profilers affect the allocation of the new objects on the processors. If the reporting frequency is low, the Resource Manager might not determine any fluctuations. If the frequency is too high, the Resource Manager might detect an increase in the load and proceed with an object migration even though the increase is only transient. Our measurements indicate that a time interval of one second is adequate to capture the load fluctuations in the processor.

The Profiler informs the Local Scheduler for each method invoked locally on the processor. Therefore, the Local Scheduler remains idle until a local method is invoked. The Local Dispatcher, which shares the data structure with the Local Scheduler selects the first method from the queue and dispatches the method for the object. The Scheduler works on the basis of hundreds of milliseconds.

We conducted two experiments to measure the CPU utilization of the Local Scheduler and Profiler, both for CPU intensive and CPU idle tasks. In the first experiment, a task involves a Camera object that continuously fetches live images from a camera and displays them. In this case, no CORBA messages are exchanged among the objects; therefore, the local Profiler measures only the CPU load. To measure the load fluctuations, due to the Camera object, we chose to run the Profiler at a higher frequency, i.e., every 100 milliseconds. The local Scheduler works continuously and uses the laxity of the task to schedule the camera object on the processor. In the second experiment, a task involves a client and server object pair, where the client makes continuous invocations on the server object. In this case, besides the load measurements, the Profiler also monitors the CORBA messages exchanged between the objects. Because there is no substantial increase in the load on the processor, the Profiler measures the usage of the CPU every second.

Figure 4 displays the overhead of our components in the first set of experiments. Figure 5 shows the same information as Figure 4 but in a different scale to illustrate better the overhead of our scheduling components. Figure 6 displays the overhead in the second set. The results show that the Local Scheduler consumes very few of the CPU cycles. They also show that, in both cases where a large number of messages is exchanged between the client object and the server object, and when the monitoring frequency of the Profiler increases, the overhead of the Profiler is below 0.31%. 
6.2. Effectiveness of Least Laxity Scheduling

We evaluated the effectiveness of the least laxity scheduling algorithm compared to the earliest deadline first (EDF) algorithm. We used the EDF algorithm because it is an optimal dynamic scheduling algorithm for single processor environments. In the experiments we measured the task miss ratio as a function of the load on the processors. The task miss ratio is the number of tasks that miss their deadlines during a single execution of each task. For the least laxity scheduling algorithm, these are the tasks with a negative laxity value.

We used a set of five tasks with different workloads that generate requests at different rates. In the experiments we included tasks with variable computation times, deadlines and laxity values. Each of the tasks consists of a variable number of objects; each object had different computation time requirements. The tasks were unithreaded, disjoint and activated independently. The objects were distributed among three processors.

Several factors contributed to the task miss ratio. The most important factor is the CPU utilization of the processors. Other factors include the number of methods on objects invoked by a task and the processors where those objects are located. The Global Scheduler, in an effort to distribute the load on the resources equally, uses the estimated queueing latency of the objects, to decide which is the most appropriate processor to host the new object. This is the processor with the minimum increase in queueing latency.

Figure 7 illustrates the miss ratio for the tasks in the system as a function of the load on the processor. At low utilizations, both algorithms ensure that the tasks complete within their deadlines. As the load increases, the EDF algorithm incurs a larger percentage of missed deadlines than the least laxity scheduling algorithm. The reason is that, earliest deadline first uses the deadline of the task to schedule the task on the processors’ resources, which remains a fixed parameter throughout the execution of the task. In contrast, least laxity scheduling uses the laxity value as the scheduling parameter. The laxity value represents a measure of urgency of the task and is dynamically adjusted as the task executes. Therefore, if a method execution is completed more quickly than was projected, the laxity of the task increases. If the task is delayed, the laxity value diminishes, and the task’s scheduling priority increases.

7. Related Work

Scheduling work reported in the literature has been dominated by hard real-time systems. However, several researchers [21] have recognized the need for scheduling solutions in soft real-time distributed systems. Jensen et al [10] 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. Gill et al [6] use a dynamic scheduling strategy that provides scheduling assurance for critical
tasks, while offering the flexibility to optimize the use of scarce resources.

Several researchers [5, 20] have evaluated scheduling solutions for real-time applications to show that least-laxity scheduling is an effective strategy for real-time distributed systems. Gupta et al [7] have investigated scheduling algorithms based on compact task graphs. Harbour et al [8] discuss the difficulty of scheduling when tasks consist of a number of subtasks, each executing at a different priority level. Saewong et al [19] use a Cooperative Scheduling Server to manage one specific controlled resource while using a controlling resource.

Many researchers have realized that efficient dynamic scheduling algorithms require knowledge of real-time task information including the task's deadline, resource requirements and worst-case computation time. Hou et al [9] implement decentralized load-sharing mechanisms to fully utilize the system resources. Brandt et al [3] use a QoS-based resource manager, called the DQM, that uses execution-level information and current system state to ensure that applications get the resources that they need to provide adequate performance. Abdelzaher [1] uses estimation theory for automated online profiling for QoS-sensitive systems and evaluates the profiling techniques in the context of an Apache Web server. Bestavros [2] uses load profiling to distribute the load among nodes so as to maximize the chances of finding a node that would satisfy the computational needs of incoming real-time tasks. Mock et al [15] employ system-wide monitoring of activities in distributed object-oriented environments and have built a tool, JewelDC, that helps to visualize the CPU scheduling of the threads that execute on behalf of the activities.

Other researchers have focused on implementing scheduling mechanisms in general-purpose operating systems. Nahrstedt et al [14] 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 work differs from ours in that their dynamic scheduler is based on a preliminary round to capture the behavior of the tasks before the actual execution starts. Wang et al [22] have implemented a general real-time scheduling framework in a real-time extension of the Linux kernel, the RED-Linux. Their framework implements many well-known scheduling algorithms. Candea et al [4] have implemented an infrastructure that allows multiple scheduling policies to coexist simultaneously in an operating system. Their system, called Vassal, allows applications to utilize scheduling algorithms tailored to their specific needs.

8. Conclusion

We have implemented a dynamic scheduling algorithm that examines the computation times, real times and resource requirements of the tasks to determine a feasible schedule for method invocations. The schedule is driven by the laxity of the tasks invoking methods on the objects and the importance that the tasks have for the system. The decisions are made on a system-wide basis because tasks are represented as distributed method invocations that have end-to-end timing and resource requirements. Our mechanism allows an invocation of an object to carry the scheduling parameters of the task from one processor to another, yielding a system-wide scheduling algorithm that requires only local computations. Although our work is based on scheduling CORBA objects, the scheduling algorithm is generic and can be used in other distributed soft real-time systems.

References


[18] Object Management Group, Dynamic Scheduling Revised Submission, orbos/00-08-12, August 2000.


