Dynamic Migration Algorithms for 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

Complex distributed object systems require dynamic migration algorithms that allocate and reallocate objects to respond to changes in the load or in the availability of the resources. In this paper we present the Cooling and Hot-Spot migration algorithms that reallocate objects when the load on a processor is high or when the latency of a task is high. The algorithms have been implemented as a feedback loop in the Eternal Resource Management System where information obtained from monitoring the behavior of the objects and the usage of the processors' resources is used to dynamically balance the load on the processors and improve the latency of the tasks. The cost of moving an object is justified by amortization over many method invocations, and constrains the rate at which objects are moved. The experimental results show that our algorithms guarantee steady flow of operation for the tasks and gracefully migrate objects from the processors when processor overloads and high task latencies are detected.

1. Introduction

Distributed object systems require dynamic allocation of the objects on the processors to respond to transient changes in the load or in the availability of the resources. Distributed applications involve objects from different classes located on multiple processors, each processor with limited processing and memory resources and shared communication resources. The structure of the applications changes dynamically; new objects are created, existing objects are deleted, and connections between objects are established and torn down or disestablished. Thus, even an optimal initial distribution may not suffice because when many objects are invoked by multiple activities that execute concurrently and asynchronously, the processing power and available bandwidth can easily become a bottleneck. Furthermore, as the system becomes larger and more complex, it becomes more difficult to predict the needs of the applications in advance, particularly because those needs are likely to change dynamically while the applications execute. An imbalance in the workload on the processors can easily delay the completion of existing tasks and result in poor system performance.

In this paper we present Cooling and Hot-Spot migration algorithms that reallocate objects when the load on a processor is high or when the latency of a task is high. Object migration provides the ability to move the execution of an object at any time from a source processor to the destination processor of the same architecture. The advantages of object migration are multi-dimensional. Object migration enables us (1) to dynamically balance the load on the resources by reallocating objects from overloaded processors to less loaded ones, (2) to improve the latency of the tasks by searching and reallocating the particular object that is causing the largest delay for the task, and, (3) to react to processor and resource faults by reallocating the objects from the faulty processors. To validate our approach, our algorithms are implemented in the Eternal Resource Management System [11]. The system is based on CORBA [15] which is a widely accepted standard for developing distributed applications over heterogeneous platforms. The experimental results show that our algorithms guarantee steady flow of operation for the tasks and gracefully migrate objects from the processors when processor overloads and high task latencies are detected.

2. The Eternal Resource Management System

The Eternal Resource Management System provides real-time and QoS guarantees for fault-tolerant soft real-time

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

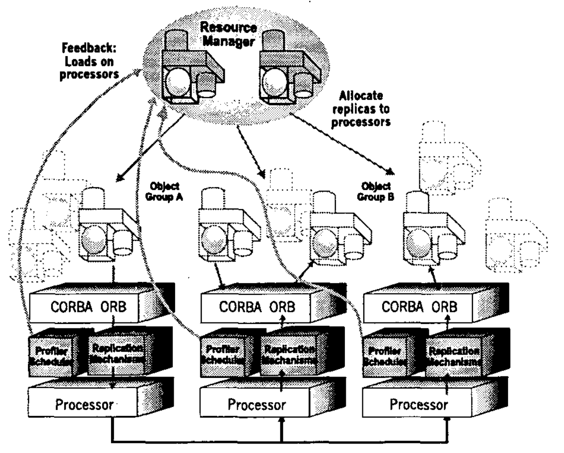

Figure 1: The Resource Management System.

distributed applications. The Resource Management System (Figure 1) is structured as a single Resource Manager, and Profilers and Schedulers located on each of the processors. The Resource Manager has a global view of the system and is responsible to distribute the application objects on the processors. The Profiler on the processor monitors the behavior of the objects on the processor and measures the current load on the processor resources. The Resource Manager maintains a current view of the application objects in the system by collecting feedback from the Profilers. The Schedulers exploit information collected by the Resource Manager to schedule tasks to meet soft real-time deadlines.

The Resource Management System works in a three-level feedback loop structure. The three-level feedback loop of the Resource Manager allows activities with different levels of temporal granularity: scheduling at the level of milliseconds, profiling over seconds, and migration over many seconds. By profiling the behavior of the application objects at run time and scheduling the method invocations across multiple processors, the Resource Manager determines when resources have become overloaded and when tasks cannot meet their deadlines, migrating objects to other processors to mitigate such conditions.

2.1. Task Parameters

An application task is modeled as a sequence of method invocations of objects distributed across multiple processors. The execution of a task is triggered by a client thread and multiple tasks invoked by different client threads can be executed concurrently. The execution times of the tasks are affected by the objects invoked by the tasks, the number of tasks in the system and the processing and communication times of the methods of the objects invoked by the tasks. Each application task t is characterized by the following parameters:

- $Deadline_t$: the time interval, starting at task initiation within which task t should be completed, specified by the application designer.
- $Importance_t$: a metric that represents the relative importance of the task, specified by the application designer.
- $Projected\_latency_t$ or ($Computation\_Time_t$): the estimated amount of time from initiation to completion of task t. The $Projected\_latency_t$ is computed from the processing times of the methods on the processors and their communication times as they are invoked by the tasks.
- $Laxity_t$: the difference between $Deadline_t$ and $Projected\_latency_t$. The laxity value represents a measure of urgency for task t and is used to schedule the methods of the objects invoked by the task.

2.2. Object Parameters

Each object is characterized by a set of methods and the percentage of resources required for the execution of the methods. With each object i, we associate the following parameters:

- $Methods_i$: the set of methods of object i.
- $Processor\_address_i$: the network address of the processor where the object i is located.
- $Object\_state_i$: the current state of the object. The object is `active` if it is currently executing and is `inactive` if it is idle or waiting for a resource.
- $Resource\_utilization_i$: the percentage of the processor's resources required for executing methods of the object.

2.3. Method Parameters

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

- $Method\_name_m$: the name of the method.
- $Object_m$: the object of which m is a method.
- $Mean\_processing\_time_m$ or (τ_{mp}): the mean time required for method m to execute locally on processor p, excluding queueing time and the time required for embedded invocations of other methods.
- $Mean\_execution\_time_m$: 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 $Mean\_execution\_time_m$ includes

queueing time but excludes the time required for embedded invocations of other methods.

- $Mean\_communication\_time_{mn}$: the mean time to communicate an invocation from method m to method n and to communicate the response back.
- $Mean\_invocations_{mn}$: the mean number of invocations that a single invocation of method m makes on method n.

3. Object Profiling

The Profiler on each processor measures the actual computation and execution times of the methods of the objects and measures the usage on the processor's resources during the executions. The accuracy of the timing estimates depends on the accuracy of measured data and the frequency of measured events. 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 where the objects are located. For example, in heavily loaded systems, the execution times are longer. The frequency of measurements depends on the application tasks and the flow of operation in the system. For example, small-scale highly critical systems require small monitoring intervals, while large-scale enterprise systems require larger monitoring intervals.

The Profiler monitors the method invocations among the CORBA objects. CORBA provides the General Inter-ORB Protocol (GIOP) and its TCP/IP implementation called the Internet Inter-ORB Protocol (IIOP) for communication among objects over heterogeneous platforms. Each method invocation or response, as monitored by the Profilers is characterized by the following information: *(Action, local_object i, invoking_method m, remote_object j, invoked_method n, time_of_invocation T)* where Action is determined by the Profilers and can be one of the following: LOCAL_START, LOCAL_COMPLETE, REMOTE_START, REMOTE_COMPLETE. The Profilers distinguish between a local invocation that does not measure any network communication time (LOCAL_START, LOCAL_COMPLETE) and a remote invocation that does include network communication time (REMOTE_START, REMOTE_COMPLETE). The Profilers attach a timestamp T to each of the method invocations and can therefore measure the number of method invocations, as well as the execution and communication times of the local and remote methods as invoked by the tasks.

When a method invocation arrives locally on the processor, the Profiler measures the local execution time for the method, that is, the time required, after receipt of a message invoking a method for that processor, to complete the invocation. This includes the processing time (τ_{mp}) of the method to execute locally on the processor and the time the method spends on the Scheduler's queue waiting for the CPU to be released. The queueing time is computed as the difference between the arrival time of the method invocation and the time the method starts executing. The order with which the methods will be executed depends on the other objects currently scheduled or queued on the processor. The local Scheduler uses the laxity value of the task invoking the method, and the importance of the object and the task to the system, to order the method in the local queue[12].

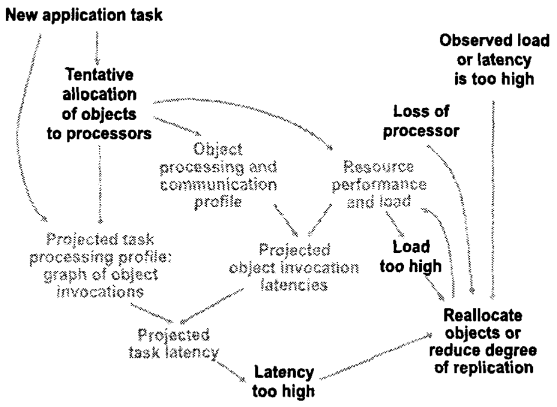
New application task
Tentative allocation of objects to processors
Observed load or latency is too high
Loss of processor
Object processing and communication profile
Resource performance and load
Projected task processing profile: graph of object invocations
Projected object invocation latencies
Load too high
Reallocate objects or reduce degree of replication
Projected task latency
Latency too high

Figure 2: Object allocations and reallocations.

During the method execution, the Profiler measures the proportion of the processing load, the amount of memory and the disk bandwidth required for executing the method locally on the processor. The Profiler obtains this information by using system calls to the `/proc` interface. The `/proc` interface contains information for each process currently running in the system. Note though, that, this is information on a per process basis, therefore the Profiler is responsible to associate this information with the objects currently running on the processor and the methods invoked.

For each method m of an object i invoked locally on the processor, the Profiler supplies feedback information to the Resource Manager which determines the further allocation and reallocation of the objects to the processors. The Resource Manager uses the Profilers' feedback to construct the Graphs of Method Invocations for the tasks, to calculate the projected latencies, and to estimate the laxities for the tasks.

4. Object Migration

The Resource Manager (Figure 2) tries to maximize the probability of satisfying the soft real-time response requirements of all the application tasks in the system, but this cannot always be achieved. In a complex evolving system, the behavior of the objects can change dynamically as the system executes. Furthermore, the addition of a new task can result in degradation of the performance of existing tasks. The Resource Manager triggers the migration of objects to different processors, or decides to reduce the degree of replication of objects [13] when the current load on a processor is too high, or when the latency of a task is too high. Object migration may also be required when a processor fails.

4.1. Cooling Algorithm

The Resource Manager determines an overloaded processor p when the $load_p$ on the processor has increased above an upper load bound HIGH. Object migration is greatly affected by the frequency with which the Profilers provide feedback information to the Resource Manager [11]. For example, if the frequency is high, the Resource Manager may determine an overloaded processor and proceed with an object migration although the increase may be only transient. Therefore, the Resource Manager ensures that this is indeed an overloaded processor and then tries to reduce the load on the processor, by migrating one or more objects from that processor to a processor with a lower load. The Resource Manager also considers any physical constraints on the objects. For example, a multimedia object that grabs live images from a camera, is tight to a specific processor and cannot be reallocated.

Once the Resource Manager has identified an overloaded processor p, it calculates the contributions to the load on processor p made by the various objects on p. The $Object\_load$ is derived from the mean processing time for the various methods of the object and the number of invocations of those methods by the application tasks per second. Given the mean processing time τ_{mp} for method m on processor p, the mean number x_{tm} of invocations of method m made by task t and the mean execution time T_t of task t, the Resource Manager computes the object load for object i as:

$$Object\_load_i = \sum_t \sum_m \frac{x_{tm} * \tau_{mp}}{T_t}$$

The objects are considered in the order of decreasing $Object\_load_i$. The Resource Manager then selects the object i with the highest $Object\_load_i$ and determines whether it can be moved to a less loaded processor q. To find the most appropriate processor to host a new object, the Resource Manager tries to maximize the probability of accepting a new object with a high load, by reserving enough resources in a single processor (if available) for such an object, without having to keep idle resources. To do that, the Resource Manager uses the best-fit allocation technique which tries to find the processor that minimizes the wasted space after hosting the new object. Note that, although object migration reduces the load on the overloaded processor,

```
Cooling_Algorithm()
  find overloaded processor $p$ with $load_p > HIGH$
  while $load_p > HIGH$
    for all objects $i$ of processor $p$ in decreasing $Object\_load_i$
      find processor $q$ to host object $i$ using best-fit allocation
      if ($Object\_load_i + load_q <$ min(HIGH,$load_p$))
        move object $i$ from processor $p$ to processor $q$
        update processor_name of object $i$
        $load_p = load_p - Object\_load_i$
        $load_q = load_q + Object\_load_i$
```

Figure 3: Pseudocode for the Cooling Algorithm.

this may create another overloaded processor in the system. The Resource Manager uses the $Object\_load_i$ to calculate the effects of increased load on the latencies of the existing tasks in the destination processor q, as the result of the migration of object i to processor q. The Resource Manager migrates the object in that processor, only if the queueing latencies of the applications in the new processor will remain less than a specific portion. The pseudocode for the Cooling algorithm is given in Figure 3.

The benefit obtained from the Cooling algorithm depends on the load on the processors, the objects in the system and the processing and communication times of the objects as tasks invoke methods on the objects. If the Resource Manager cannot find a processor to move the object, this indicates that all processors are overloaded and the object migration will not provide any benefit to the system. In this case, the Resource Manager uses the importance metric of the tasks to decide which task to remove from the system. The Resource Manager does not try to migrate short-lived objects, because the migration will not produce a noticeable benefit to the system. The cost of moving an object is justified with amortization over many method invocations.

4.2. Hot Spot Algorithm

As each application task executes, the Resource Manager estimates the projected latency for the task t, to check whether the latency of any task is too high a proportion of its deadline. Given that ρ_p is the load on processor p, τ_{mp} represents the mean processing time for method m on processor p, σ_{mc} is the mean transmission time for invoking method m on processor p and x_{tm} is the mean number of invocations of method m made by task t, the Resource Manager computes the projected latency for the entire task as:

$$Projected\_latency_t \equiv \sum_{m,p:m\epsilon p} \left\{ \frac{x_{tm}\tau_{mp}}{1-\rho_p} + \frac{x_{tm}\sigma_{mc}}{1-\rho_c} \right\}$$

where $m \epsilon p$ denotes that the object i of which m is a

```
Hot_Spot_Algorithm()
  for all processors $p$
    for all objects $i$ of processor $p$
      compute $Queueing\_latency_{tip}$
  for all objects $i$ of processor $p$
    find processor $q$ to host object $i$ using best-fit allocation
    in decreasing order of $Queueing\_latency_{tip}$
    if $Object\_load_i + load_q <$ min(HIGH, $load_p$)
      move object $i$ from processor $p$ to processor $q$
      update processor_name of object $i$
      break
```

Figure 4: Pseudocode for the Hot Spot Algorithm.

method executes on processor p. The Resource Manager's estimate of the projected latency for the task t is based on the reports of the Profilers.

Then, the Resource Manager searches for the object that is causing the largest delay for task t. The Resource Manager estimates the $Queueing\_latency$ of all the objects i invoked by task t on all processors p as:

$$\begin{aligned} Queueing\_latency_{tip} &\equiv \sum_{m\epsilon i} \frac{x_{tm}\tau_{mp}}{(1-\rho_p)} - x_{tm}\tau_{mp} \\ &= \sum_{m\epsilon i} \rho_p \frac{x_{tm}\tau_{mp}}{(1-\rho_p)} \end{aligned}$$

The largest delay for task t is caused by the object i, executing on processor p, whose methods m cause, in aggregate, the largest increase in the latency to the completion of task t because of queueing, computed as:

$$Queueing\_ratio_i = \frac{Queueing\_latency_{tip}}{Laxity_t}$$

The Resource Manager identifies the object i that is causing the largest queueing delay for task t and the processor p on which object i is executing. Then, it uses the best-fit allocation technique to select a processor q and attempts to move object i to processor q. As in the Cooling algorithm, the Resource Manager ensures that the object migration does not result in a new overloaded processor. Alternatively, the Resource Manager tries to move another object from processor p to processor q. The above process continues until the Resource Manager determines an appropriate object to move. The pseudocode for the Hot Spot Algorithm is given in Figure 4.

Different scheduling mechanisms have a different effect on the benefits of migration. Our scheduling algorithm assigns higher priorities to the most urgent and most important tasks in the system. Therefore, even if the execution of a task is delayed, its laxity value will diminish and its scheduling priority will increase. When the task t completes, the remaining laxity of the task is recorded as $Residual\_Laxity_t$. The Resource Manager compares the *Residual Laxity* with the *Initial Laxity* of the task, as was estimated by the Resource Manager and measured by the Profilers during the task's previous executions. If the Resource Manager's estimate for the projected latency for task t is accurate, the residual laxity when task t completes will be the same as the initial laxity for task t. The Resource Manager uses the ratio

$$\frac{Residual\_Laxity_t}{Initial\_Laxity_t}$$

to adjust the estimates of the projected latency for the task. For example, if the residual laxity is frequently smaller 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 Resource Manager updates the task's projected latency by averaging the previous estimate with the latest measurement.

The Hot-Spot algorithm offers more benefit to tasks that invoke methods of objects on multiple processors, where the queueing delay caused by an invoking object on a processor can cause the task to miss its deadline. Migrating such an object, helps not only the migrant object but also the other objects on the processor. Thus, identifying objects with high $Queueing\_latency$ is critical to the performance of all tasks in the system. To avoid unnecessary object migrations, the Resource Manager estimates the remaining time for the task to complete. If the remaining time is small, object migration will not increase the performance gain since the migration overhead will outweigh the benefit of balancing the remaining load.

5. Transferring Object State

The overall problem in migration is to maintain the state of the object even after the migration. In practice, to transfer the state of an object, the state must be extracted from its environment on the original processor, transmitted to the destination processor, and reinstantiated in the object's new environment. The new object must have exactly the same behavior with the original object, the same virtual memory and the same access to any system resources that it had previously. Furthermore, the object migration must be undetectable to the invoking objects in the system.

In our current implementation, we have focused on migrating objects with a single thread of control. The state of unithreaded objects typically consists of text region, data region and the heap. The text region is the code region in the process's virtual address space that is typically memory mapped from the executable from the disk. The data region consists of global and static variables used by the program. The heap contains the dynamic memory allocated for the object. Information about the heap, such as its starting address and size, can be obtained through system calls to the /proc interface. The time to migrate the object is proportional to the amount of time to transfer its virtual memory to the destination machine.

An object may also use private communication channels, such as files, TCP/IP sockets and shared memory. This state is more difficult to transfer because it involves state maintained by the infrastructure on which the object executes. For example, the state of an open file includes the internal identifiers for the file, the current access position, and possibly cached file blocks. When the process migrates, all the handles that are opened become invalid on the new machine.

This state transfer can be achieved by using interception mechanisms to monitor the external effects of the application and record events such as the creation of sockets and the opening of files. For example, Nasika *et al* [14] inject a wrapper DLL into an application at run-time that manages the execution of the application. This can be done transparently, without modification to the application or the operating system.

When the object migrates, its state should be restored so that it continues to execute as before. This is done by creating a new object at the destination machine and manipulating its state so that it is identical to that of the original object. However, other state of the object may not be meaningful to transfer. Such state is usually associated with physical devices on the source machine.

The transfer of the object's state averages from hundreds of milliseconds to migrate an object with a small state to many seconds to migrate an object with a large state. To migrate an object, the object must be currently quiescent. Using the object profiles constructed from the Profilers' feedback, the Resource Manager can accurately predict when the objects will be invoked again. To ensure that the object remains quiescent during the migration, the Scheduler can reschedule the invocations or responses of that object. This is necessary to make sure that the state of the object does not change during the migration process.

6. Experimental Results

We have investigated the effectiveness of our Cooling and Hot-Spot migration algorithms by considering the benefit obtained when the load on a processor is high or the latency of a task is high.

The experimental platform for our measurements consisted of eight 167 MHz Sun ULTRASparcs running Solaris 2.5.1 with the VisiBroker ORB 3.3 over 100 Mbit/s Ethernet. We used ten application tasks, each consisting of three to five objects. The application tasks were not disjoint, as different tasks could invoke methods on the same objects. All objects were implemented using the C++ language and

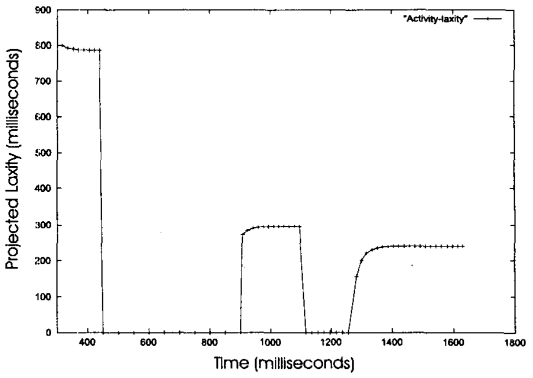

Figure 5: The projected laxity value of the task as the task invokes methods on objects on multiple processors, over multiple invocations of the task.

were unithreaded. The objects were scheduled using the realtime scheduling class of the Solaris operating system.

Figure 5 shows a task that executes every 20 milliseconds across multiple processors. The projected laxity is the Resource Manager's estimate for the laxity of the task when the task is launched. Twice in its execution, the system becomes overloaded, the projected laxity of the task becomes negative and the task cannot be executed. Each time, an object is migrated and the task resumes execution.

When the task starts executing, there is some adjustment in the laxity value as can be seen in Figure 5. The reason is that the first time the task executes, the Resource Manager uses the computation time of the task estimated by the user to compute the laxity value of the task. As the task executes, the Profilers monitor the actual execution times of the methods invoked by the task; therefore the Resource Manager can estimate the laxity value of the task with great accuracy. A similar refinement occurs after object migration and also after a major change in load. As the task is delayed, its laxity value diminishes, so next time the task executes it has a smaller laxity value and a higher priority.

As an application task invokes methods on objects on multiple processors, our objective is to maximize the probability that the laxity value of the task will be sufficient to guarantee that the task will meet its deadline. Therefore, the Resource Manager tries to improve the queueing times of the methods locally on the processors.

Figure 6 shows the effectiveness of our Cooling algorithm in the laxity value of a task. The bottom graph in the figure represents the *Residual Laxity* of the task that is measured by the Profilers when the task finishes execution. The top graph in the figure represents the projected laxity of the task, that is estimated by the Resource Manager based on the Profilers' feedback during the previous executions of the task. The figure shows that, as the task executes, the Resource Manager determines that a processor is overloaded

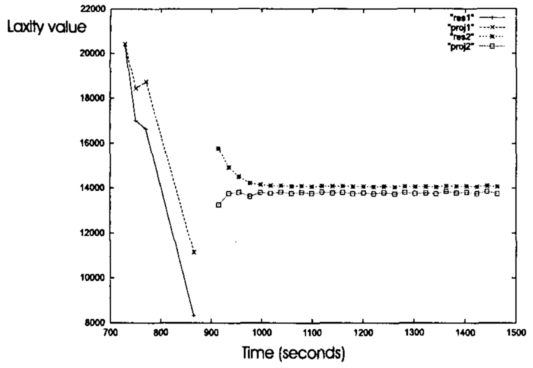

Figure 6: The Cooling algorithm reallocates objects when the load on a processor is high.

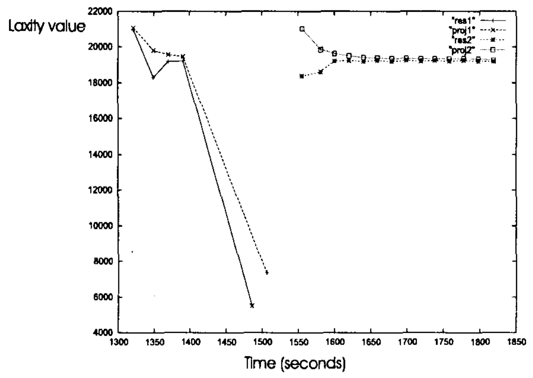

Figure 7: The Hot-Spot algorithm reallocates objects when the latency of a task is high.

(which causes the decline in projected and residual laxity) and chooses to migrate one of the objects invoked by this task. When the task starts executing again, its laxity value is larger.

Figure 7 shows the effectiveness of our Hot-Spot algorithm in the laxity value of the task. The figure shows that the marginal improvement on the laxity value for the Hot-Spot algorithm is significantly greater than the Cooling algorithm, because the Resource Manager migrates that particular object that is contributing most to the delay of the task. The figure shows that the benefit obtained from the Hot-Spot algorithm is greater. The reason is that the Hot-Spot algorithm migrates the specific object that is causing the largest delay to the latency of the task because of queueing, to guarantee steady flow of operation for the tasks.

The start time, and thus the decision time, depends on the size of the state of the object. The cost of migrating an object must be justified by amortization over many invocations. This constrains the rate at which objects are moved. It is inappropriate to migrate an object immediately, because migration is costly and the overhead may be transient.

7. Related Work

The topic of allocating or reallocating tasks, objects or resources to processors in distributed systems has been extensively studied in the literature. Rajkumar et al [17] focus on allocating multiple types of resources to multiple applications so that system objectives are maximized. Deplanche *et al* [4] have studied task migration in the context of reconfiguration in fault-tolerant real-time distributed systems. In contrast to our soft real-time scheduling algorithm, they focus on satisfying hard real-time scheduling constraints.

Many researchers [9] have realized the need to profile the application tasks as the behavior of the application objects can change dynamically. Baxter *et al* [1] describe the dynamic profiling of tasks in a parallel processor system, so that the initial allocation can be refined and tasks can be migrated to other, more appropriate, processors. Rabinovich et al [16] propose algorithms that dynamically replicate and migrate Internet objects in response to demand changes.

Several researchers [3] have shown that object migration is a useful mechanism to optimize distributed monolithic systems. Douglis *et al* [6] have implemented process migration to offload work onto idle machines in the Sprite operating system. Steketee *et al* [18] describe their experiences with the implementation of a process migration mechanism in the Amoeba distributed operating system. They have shown that the speed of process migration is limited only by the throughput of the network adapters used in the configuration, and that the overhead is comparable to that of process creation.

Several researchers [5, 7, 10] recognize that efficient dynamic scheduling algorithms require that the allocation must be closely related to scheduling and devise scheduling algorithms to balance the load on multiprocessor environments. Bettati et al [2] propose a dynamic resource migration scheme that dynamically changes resource allocations (migrate resources from node-to-node in the network) without affecting the timing guarantees provided to the clients by the active real-time connections. Similar to our work, Hou *et al* [8] have used the minimum-laxity-first-served algorithm to study task migration to meet real-time deadlines.

8. Conclusions

We have developed Cooling and Hot-Spot migration algorithms that allocate and migrate objects in the Eternal Resource Management System. By profiling the behavior of the application objects at run time and scheduling the method invocations across multiple processors, the Resource Manager determines when resources have become overloaded and when tasks cannot meet their deadlines. The migration algorithms allow the system to balance the load across multiple processors and to respond to transient changes in the load or in the availability of the resources.

References

[1] J. Baxter and J. H. Patel, "Profiling based task migration," *Proceedings of the IEEE 6th International Parallel Processing Symposium*, Beverly Hills, CA (March 1992), pp. 192-195.

[2] R. Bettati, A. Gupta, "Dynamic resource migration for multiparty real-time communication," *Proceedings of the IEEE 16th International Conference on Distributed Computing Systems*, Hong Kong (May 1996), pp. 646-655.

[3] O. Ciupke, D. Kottmann, H.-D. Walter, "Object migration in non-monolithic distributed applications," *Proceedings of the IEEE 16th International Conference on Distributed Computing Systems*, Hong Kong (May 1996), pp. 529-536.

[4] A. M. Deplanche and J. P. Elloy, "Task redistribution with allocation constraints in a fault-tolerant real-time multiprocessor system," *Proceedings of the IFIP WWG 10.3 Working Conference on Distributed Processing*, Amsterdam, Netherlands (October 1987), pp. 133–150.

[5] M. L. Dertouzos and A. K.-L. Mok, "Multiprocessor on-line scheduling of hard real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12 (December 1989), pp. 1497-1506.

[6] F. Douglis, J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software-Practice and Experience*, 21(8) (August 1991), pp. 757-785.

[7] R. Gupta, D. Mosse and R. Suchoza, "Real-time scheduling using compact task graphs," *Proceedings of the IEEE 16th International Conference on Distributed Computing Systems*, Hong Kong (May 1996), pp. 55–62.

[8] C. J. Hou and K. G. Shin, "Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems," *IEEE Transactions on Computers*, vol. 43, no. 9 (September 1994), pp. 1076-1090.

[9] J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao and R. Bettati, "RT-ARM: A real-time adaptive resource management system for distributed mission-critical applications," *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems*, San Fransisco, CA (December 1997).

[10] E. D. Jensen, C. D. Locke and H. Tokuda, "A time-driven scheduling model for real-time operating systems," *Proceedings of the IEEE Sixth Real-Time Systems Symposium*, San Diego, CA (December 1985), pp. 112-122.

[11] V. Kalogeraki, L. E. Moser and P. M. Melliar-Smith, "Using multiple feedback loops for object profiling, scheduling and migration in soft real-time distributed object systems," *Proceedings of the IEEE Second International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint Malo, France (May 1999), pp. 291-300.

[12] V. Kalogeraki, P.M. Melliar-Smith and L.E. Moser, "Dynamic scheduling for soft real-time distributed object systems," *Proceedings of the IEEE Third International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport, CA (March 2000), pp. 114-121.

[13] V. Kalogeraki, P.M. Melliar-Smith and L.E. Moser, "Managing object groups in fault-tolerant distributed object systems," *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada (August 2000), pp. 509-516.

[14] R. Nasika, P. Dasgupta, "Transparent Migration of Distributed Communication Processes," *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada (August 2000), pp. 380-386.

[15] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, formal/00-10-01 Version 2.4, October 2000.

[16] M. Rabinovich, I. Rabinovich, R. Rajaraman, A. Aggarwal, "A dynamic object replication and migration protocol for an Internet hosting service," *Proceedings of the IEEE 19th International Conference on Distributed Computing Systems*, Austin, TX, (May-June 1999), pp. 101-113.

[17] R. Rajkumar, C. Lee, J. Lehoczky and D. Siewiorek, "A resource allocation model for QoS management," *Proceedings of the IEEE 18th Real-Time Systems Symposium*, San Francisco, CA (December 1997), pp. 298-307.

[18] C. Steketee, P. Socko, B. Kiepuszewski, "Experiences with the implementation of a process migration mechanism for Amoeba," *Proceedings of the Nineteenth Australasian Computer Science Conference*, Melbourne, Australia, (January-February 1996), pp. 140-148.