In Proceedings of ICDCS'97: The IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997.

Load Profiling

A Methodology for Scheduling Real-Time Tasks in a Distributed System*

Azer Bestavros
(best@cs.bu.edu)
Computer Science Department
Boston University, MA 02215

Abstract

Traditionally, the goal of load management protocols for distributed systems has been to ensure that nodes are equally loaded. In this paper, we show that for real-time systems, load balancing is not desirable since it results in the available bandwidth being distributed equally amongst all nodes—in effect making all nodes in the system capable of offering almost the same bandwidth (e.g., in cycles per second) to incoming tasks. We show that this "one size fits all" practice leads to a higher rate of missed deadlines as incoming tasks may be denied service because they require bandwidth that cannot be granted at any single node—while plenty of fragmented bandwidth is collectively available in the system. We propose a new load-profiling strategy that allows the nodes of a distributed system to be unequally loaded so as to maximize the chances of finding a node that would satisfy the computational needs of incoming real-time tasks. The performance of the proposed protocol is evaluated via simulation, and is contrasted to other dynamic scheduling protocols for real-time distributed systems.

1. Introduction

Loosely coupled, time-critical distributed systems are used to control physical processes in complex applications, such as controllers in aviation systems and nuclear power plants. If missing a task's deadline is catastrophic, then the task's deadline is considered to be *hard*, and the task is categorized as *critical*, otherwise the task's deadline is considered to be *firm* or *soft*, and the task is categorized as *essential*. If missing a deadline implies that the task can/must be discarded, then the dealdine is termed *firm*. However, if a task must be executed *even* after its deadline is missed, then the deadline is called *soft*. Finishing the execution of a task past its soft deadline is necessary to avoid incuring a penalty. In this paper, we consider only hard and firm deadlines for critical and essential tasks, respectively.

To guarantee that critical tasks will never miss their deadlines, their characteristics must be known in advance and accordingly, their resource requirements must be preallocated in advance. To allow such preallocation, critical tasks are treated as *periodic* processes, where the period of the process is related to the maximum frequency with which the execution of this process is requested. This assumption of periodicity is wasteful of system's resources, since it is based on a worst case that may rarely materialize. Fortunately, most tasks in a real-time system are essential (*i.e.* not critical); and since meeting the deadlines of these tasks does not have to be guaranteed *a priori*, real-time systems often use a *best-effort* scheduling approach for essential tasks. In particular, the characteristics of these tasks are *not* assumed to be known *a priori*, and requests for executing these tasks are *not* assumed to be periodic in nature, but rather *sporadic*.

For a distributed, multiprocessor environment, scheduling is an NP-hard problem [5] and requires *a priori* knowledge of task deadlines, computation times and start times [3]. The difficulty of scheduling in a real-time multiprocessor system is further exacerbated by the synchronization problems of loosely coupled distributed systems. Accordingly, techniques devised for such systems are best described as heuristics based on *load-shedding* approaches that attempt to balance the system load amongst the different nodes therein [6]. A set of such heuristics, including *focused addressing* and *bidding*, are described in [15, 10]. Using the focused addressing heuristic, a sporadic task, whose deadline cannot be met by executing it locally, is sent to another node, called the *focused node*, that is estimated to have sufficient surplus of cycles to complete the task before its deadline. Using the bidding heuristic, when

*This work has been partially supported by the NSF (grant CCR-9308344).

a node fails to schedule a sporadic task locally, it asks for *"bids"* from the rest of the nodes in the system, and depending on the received bids it selects one of them as the target node. In [10], a *flexible* heuristic that combines focused addressing and bidding is also proposed. Using that heuristic, if a node cannot be found via focused addressing, the bidding scheme is invoked (in fact, the bidding scheme is invoked while communication with the focused node is in progress). Spring [11, 16] is an example of a multi-processor system that supports scheduling for real-time sporadic tasks.

In [18], *load balancing* was found to reduce significantly the mean and standard deviation of job response times, especially under heavy or unbalanced workload. For non-real-time systems, reducing the mean and standard deviation of job response times is an appropriate measure of performance. However, for real-time systems, such a measure may be completely misleading. To explain this dichotomy, it suffices to point out that in real-time systems, the metric of interest is *not* response time, but the percentage of tasks that are completed *before* their deadlines.

In this paper, we present and evaluate a decentralized algorithm for scheduling sporadic tasks on a loosely-coupled distributed system in the presence of other critical, periodic tasks. The main contribution of our work is the introduction of the load-profiling concept and the establishment of its superiority for real-time systems.

2. Load Profiling

System Model and Assumptions: We model a distributed real-time system as a set of nodes connected via a communication network. Each node consists of two processors: one is dedicated to the execution of critical and essential tasks and the other is dedicated to the execution of system tasks, such as admission control protocols, scheduling protocols, communication functions, among others. The allocation of system and application tasks to two (or more) separate processors is typical in real-time environments because it prevents the unpredictability associated with system management functions (*e.g.,* interrupts from I/O devices) from affecting the execution of time-critical tasks.

Each node in the system is associated with a (possibly empty) set of critical, periodic tasks, which possess hard execution deadlines. We assume that the deadline of a periodic task is the beginning of the next period. Thus, a periodic task can be described by the pair (C_i, P_i), where C_i is the required execution time each period P_i. The characteristics of periodic tasks are known *a priori*. This enables them to be scheduled off-line during system startup using algorithms for scheduling periodic tasks, such as RMS [7].

In addition to periodic tasks, sporadic tasks with firm deadlines may be submitted to the system dynamically. We describe a sporadic task by the triplet (A_j, C_j, D_j), where A_j is the arrival time of the task (*i.e.* the time at which the task was submitted for execution), C_j is the execution time necessary to complete the task, and D_j is the deadline of the task. The characteristics of a sporadic task are not known *a priori*; they become known when the task is submitted for execution. Upon submission, the node tries to schedule the sporadic task locally using algorithms for scheduling sporadic tasks on a single processor [12, 2, 17]. If not successful, the task is forwarded for remote execution on a different node.

For a given sporadic task, we define the *time-to-live* for a sporadic task as the difference between its deadline and its arrival time. The ratio between a task's execution time and its time-to-live defines the *utilization requirement* (ρ_j) for that task, where $\rho_j = C_j/(D_j - A_j)$. A ρ_j value close to 1 is indicative of a task that requires almost 100% of the CPU cycles available at a node. A ρ_j value close to 0 is indicative of a task that requires only a small percentage of the CPU cycles available at a node. The difference between the time-to-live and the execution time of a task define its *laxity*. We define the *laxity ratio* to be the ratio $\frac{(D_j - A_j - C_j)}{C_j} = \frac{(1-\rho_j)}{\rho_j}$. The characteristics of individual sporadic tasks are not known until these tasks are submitted for execution. However, we assume that the distribution of ρ_j is known *a priori*, or else it could be estimated dynamically.

Local Scheduling Algorithms: For scheduling periodic tasks, we use the Earliest Deadline First (EDF) algorithm—a dynamic, preemptive scheduling algorithm. For a given task set $\mathcal{T}$, with n periodic tasks, a necessary and sufficient condition for the EDF to feasibly schedule the task set, is $U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$. Since the characteristics of the periodic tasks are known a priori, we can guarantee their schedulability by simply computing the *utilization factor* U, during the system setup.

For scheduling sporadic tasks locally, we use the results obtained in [1]. Two implementations of the EDF, called EDS and EDL, are possible such that tasks are processed as soon as possible and as late as possible, respectively. Following the notation in [1], we introduce the availability function $f_Y^x(t)$, with respect to a task set Y, scheduled according to the scheduling algorithm x in the time interval $[0, t]$, to be:

$$f_Y^x(t) = \begin{cases} 1 & \text{if the processor is idle at } t \\ 0 & \text{otherwise.} \end{cases}$$

For any time instances t_1 and t_2, the integral $\Omega_Y^x(t_1, t_2) = \int_{t_1}^{t_2} f_Y^x(t)dt$ gives the total number of units of time the processor is idle in the interval $[t_1, t_2]$. Because of the cyclicity property of earliest-deadline protocols, for a periodic task set $\mathcal{T}$ consisting of n tasks, and for any instant t we have $f_{\mathcal{T}}^{ED}(t) = f_{\mathcal{T}}^{ED}(t+kP), k \geq 1$, where $P = lcm(P_1, P_2, \cdots, P_n)$, and P_i is the period of task $i \in \mathcal{T}$. So, over the time interval $[0, P]$, and consequently any window of time thereafter, the remaining processor idle time is known, by computing the previous function. For a sporadic task set $\mathcal{S}$, with D as the maximum deadline of the sporadic tasks in $\mathcal{S}$, it holds that for any instant $t \leq D$, $\Omega_{\mathcal{T}}^{\mathrm{EDS}}(0,t) \leq \Omega_{\mathcal{T}}^{x}(0,t)$, and $\Omega_{\mathcal{T}}^{\mathrm{EDL}}(0,t) \geq \Omega_{\mathcal{T}}^{x}(0,t)$, where x is any preemptive scheduling algorithm. Thus, scheduling tasks by EDL will provide us with the largest number of idle processor cycles over the interval $[0, t]$.

In order to check the schedulability of a sporadic task on a local node, we have implemented an algorithm, `LSCHED`, that utilizes the above results. `LSCHED` is invoked whenever a sporadic task arrives at a node. It looks ahead in time and decides whether the sporadic task can be accepted locally using EDL and be guaranteed enough cycles to finish before its deadline. `LSCHED` runs in time linear with respect to the number of tasks accepted locally.

Remote Scheduling of Sporadic Tasks: Following the terminology in [4], our algorithm for scheduling aperiodic tasks is composed of two components: a *transfer policy* and a *location policy*. Our transfer policy is to forward a sporadic task to another node if the amount of idle processor time until the task's deadline is less than the task computational requirements (*i.e.* if scheduling the sporadic task locally fails). Otherwise, the task is guaranteed execution on the node to which it was initially assigned. The task transfer decision is made dynamically and is based on the current state of the node and the characteristics of the task. The location policy dictates the way the target node is selected. This selection is made in such a way so as to maximize the probability that the chosen target will indeed be capable of honoring the execution requirements of the transferred sporadic task. This is done through the introduction of *load profiling*, which we discuss next.

Load Profiling vs Load Balancing: Consider a system with N identical nodes. Let $f(u)$ denote the probability that the utilization requirement of a submitted sporadic task will be u, where $0 \leq u \leq 1$. Let W denote the overall load of the system, expressed as the sum of the utilization over all nodes (*i.e.* $N \geq W \geq 0$). A load-balanced system would tend to distribute this load equally amongst all nodes, making the utilization at each node as close as possible W/N. A load-profiled system would tend to distribute this load in such a way that the probability of satisfying the utilization requirements of incoming tasks is maximized.

Let $\mathcal{S}$ denote the set of nodes in the system. For distributed scheduling purposes, we assume the availability of a *location policy* [4] that allows a scheduler to select a subset of nodes from $\mathcal{S}$ that are *believed* to be be capable of satisfying the utilization requirement u of an incoming sporadic task. We denote this *candidate set* by $\mathcal{C}$. Let $l_{\mathcal{C}}(u)$ denote the fraction of nodes in $\mathcal{C}$, whose available (*i.e.* unused) utilization is equal to u. Thus, $L_{\mathcal{C}}(u) = \int_0^u l_{\mathcal{C}}(u)du$ could be thought of as the (cumulative) probability that the available utilization at a node selected at random from $\mathcal{C}$ will be less than or equal to u. Thus, the probability that a sporadic task will be schedulable at a node selected randomly out of $\mathcal{C}$ is given by

$$P = \int_0^1 f(u)(1 - L_{\mathcal{C}}(u))du \qquad (1)$$

In a perfectly load-balanced system, any candidate set of nodes will be identical in terms of its utilization profile to the set of all nodes in the system. Thus, in a load-balanced system $L_{\mathcal{C}}(u) = L_{\mathcal{S}}(u) = L(u)$. Moreover, $L(u) = 1$ for $0 \leq u < (1 - W/N)$ and $L(u) = 0$ for $(1 - W/N) \leq u \leq 1$. Thus, the probability that a sporadic task will be accepted is given by $P = \int_0^{(1-W/N)} f(u)1.du = F(1 - W/N)$, where $F(u)$ is the cumulative probability function corresponding to $f(u)$. Moreover, the probability that a sporadic task will be schedulable after k trials is given by

$$P_k = 1 - (1-P)^k = 1 - F(1 - W/N)^k \qquad (2)$$

A load-profiling algorithm would attempt to *shape* the distribution of available utilization in the system $L_{\mathcal{S}}(u)$ in such a way that the choice of a candidate set $\mathcal{C}$ would result in minimizing the value of $L_{\mathcal{C}}(u)$, thus maximizing the value of P in equation 1 subject to the boundary constraint $\int_0^1 u l_{\mathcal{S}}(u)du = (1 - W/N)$. One solution to this optimization problem is for $l_{\mathcal{S}}(u)$ to be chosen as $l_{\mathcal{S}}(u) = (W/N)u_0(0) + (1 - W/N)u_0(1)$ where $v.u_0(x)$ is an impulse function of magnitude v applied at $u = x$. This solution corresponds to a system that schedules its load using the minimal possible number of nodes. Thus, a fraction W/N of the nodes in the system are 100% utilized, and thus have *no* extra cycles to spare, whereas a fraction $(1 - W/N)$ of the nodes in the system are 100% idle, and thus able to service sporadic tasks with *any* utilization requirements. The choice of any candidate set $\mathcal{C}$ from the set of idle nodes would result in $L_{\mathcal{C}}(u)$ being a step function given by:

$$L_{\mathcal{C}}(u) = \begin{cases} 0 & \text{if } 0 \leq u < 1 \\ 1 & \text{if } u = 1 \end{cases} \qquad (3)$$

Plugging these values into equation 1, we get $P = \int_0^1 f(u)(1-0)du = 1$, which is obviously optimal.

Since the *perfect fit* implied in equation 3 is known to be NP-hard, heuristics such as *first-fit* or *best-fit* are usually employed for on-line scheduling. Asymptotically, both the first-fit and best-fit heuristics are known to be optimal [8]. However, for a small value of N—which is likely to be the case in most distributed systems—best-fit outperforms first-fit.

To quantify the benefits of load profiling versus load balancing, we performed a number of simulations to compare the schedulability of sporadic tasks under two task allocation strategies. The first is a *load-balancing* strategy, whereby a task is assigned to the least utilized node out of all the nodes capable of satisfying the utilization requirements of that task. If none exist, then the task is deemed unschedulable in a load-balanced system. The second is a *load-profiling* strategy, whereby a task is assigned to the most utilized node (*i.e.* the node that provides the best fit) out of all nodes capable of satisfying the utilization requirements of that task. If none exist, then the task is deemed unschedulable in a load-profiled system. Sporadic tasks were continually generated so as to keep the overall utilization of the system (W) at a constant level. For each one of these strategies, the percentage of sporadic tasks successfully scheduled—and consequently successfully meeting their deadlines—is computed. We call this metric the *Guarantee Ratio* (G).

Figure 1 shows example results from our simulations. These results suggest that as the utilization of the system increases, the performance of both load balancing and load profiling degrades as evidenced by the lower guarantee ratio. However, the degradation for load balancing starts much earlier than for load profiling. This is to be expected, since the availability profile in a load-balanced system is not as diverse as that in a load-profiled system. Figure 1 also shows that the advantage from using load profiling is much more pronounced when the size of the system is small.

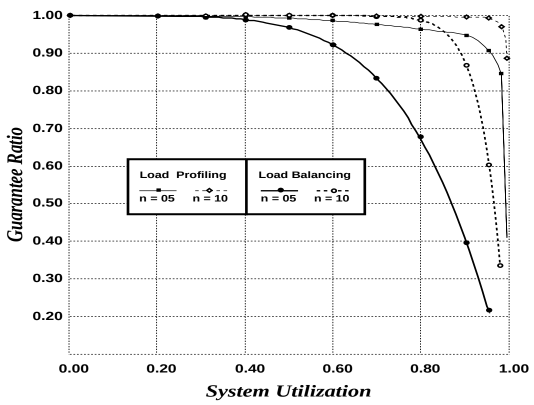

Figure 1. Profiling vs Balancing

Distributed Load-Profiling: The simulations in figure 1 assumed the existence of an oracle—a centralized scheduler possessing perfect knowledge about the utilization of all the nodes in the system. In a distributed system, the function of such an oracle must be approximated using a distributed protocol that allows nodes to exchange information about their local utilization in order to enable them to construct a global (albeit approximate) view of the overall system profile. In that respect, the most important information a node must exchange with other nodes is the localization and duration of the node's idle times and the time interval for which this information was computed. The information about idle times changes whenever a sporadic task arrives at a node and is accepted for execution. In this case, by invoking algorithm `LSCHED` the node is able to compute the new localization and duration of the idle times.

Changes in the workload of a node are signaled to other nodes based on changes in the utilization factor ρ. We define three threshold values for ρ, namely ρ_l, ρ_m and ρ_h, corresponding to four states: *lightly-loaded*, *moderately-loaded*, *heavily-loaded*, and *overloaded* systems. When the utilization of a node crosses one of these thresholds, the node sends out the localization and duration of the node's idle times and the time interval for which this information was computed to a *small subset* of nodes. To ensure that this information eventually propagates to *all* nodes in the system, we introduce a *gossiping* protocol. Using that protocol, when a certain period of time ellapses without a significant change in the load condition of a node, the node is required to initiate a *gossip session* with its neighbors. During this session, it exchanges information about its own workload and about the workload of all the other nodes in the system with its neighbors. A node that receives information about another node checks if the information received is newer than the one already kept. If this is the case, it updates its information table.

To implement the above exchange of load information, we associate with every node in the system three tasks: `PROFILE`, `MULTICAST`, and `GOSSIP`. `PROFILE` is invoked whenever the workload on the node is to be evaluated, which is typically the case when a new sporadic task is accepted or an already accepted sporadic task is completed. `PROFILE` computes the workload on the node and stores that information in appropriate data structures. `GOSSIP` is invoked whenever the workload at the node changes (*e.g.,* after `PROFILE` is invoked). Otherwise, it is invoked at least once every *GossipDe-*

lay units of time. GOSSIP sends the most up-to-date local and global workload information only to *neighboring* nodes. MULTICAST is invoked whenever the workload at the node changes considerably (*i.e.* the utilization threshold is crossed), in which case the local workload profile at the node is sent to a subset *MulticastSet* of all the nodes in the system. *GossipDelay* and *MulticastSet* are chosen in such a way that the dissemination of major workload changes is guaranteed to propagate fast enough using both MULTICAST and GOSSIP. This is necessary to ensure stability [13]. Generally speaking, by reducing the value of *GossipDelay* (*i.e.* by gossiping frequently), the size of *MulticastSet* is reduced.

Location Policy: When a node has to select a target for a sporadic task that it cannot accomodate, it does so based on its view of the workload information at other nodes in the system. First, a set (*CandidateSet*) of target nodes that are likely to accept that task is identified. This identification is based on a prediction scheme used by the sender of the task to estimate the idle cycles (at the target) until the task's deadline. If *CandidateSet* is empty, then the task is kept for a later re-submission. Next, one node from *CandidateSet* is chosen and the task is transferred to that node.

In a distributed environment, the performance of best-fit is severely affected by the inaccuracy of the workload information. The inadequacy of best-fit in a distributed environment could be explained by noting that the best-fit heuristic is the *most* susceptible of all heuristics to even minor inaccuracies in workload information. This is due to best-fit's minimization of the slack at the target node—a minimal slack translates to a minimal tolerance for imprecision. Thus, in our protocol, the process of choosing a target node out of the *CandidateSet* is carried out by a task LOCATE so as to maximize the probability of the transferred task being accepted, while maintaining the desired variability in utilization.

The probability of picking a node from *CandidateSet* is adjusted in such a way that the *availability profile*—the spectrum of available free cycles in the system—is maintained as close as possible to the expected profile of incoming time-constrained sporadic tasks. Figure 2 illustrates this idea. It shows two availability profile distributions. The first is the current availability profile of the system, which is constructed by computing the percentage of nodes in the system with *available* (*i.e.* unused) utilization larger than a particular range. The second is the desired availability profile, which is constructed by matching the characteristics of sporadic tasks—namely, the distribution of average number of CPU cycles per second needed by a sporadic task to meet its deadline. From these two availability profiles, a probability density function is constructed for the *CandidateSet*, and a node from that set is probabilistically chosen according to that density function.

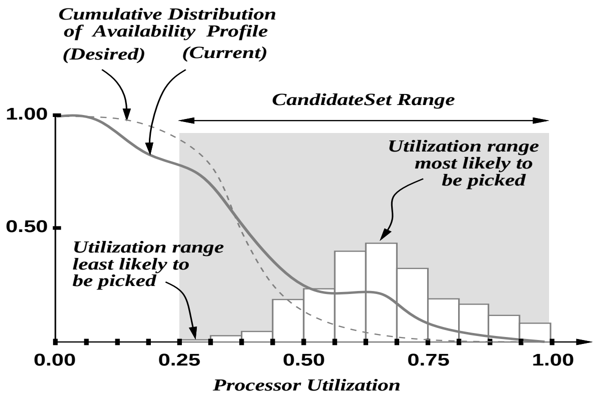

Figure 2. Maintaining a load profile

Summary of Protocol Components: Based on the above presentation, the various tasks involved in our protocol on each node in the system—as well as the flow of information between these tasks—are shown in figure 3.

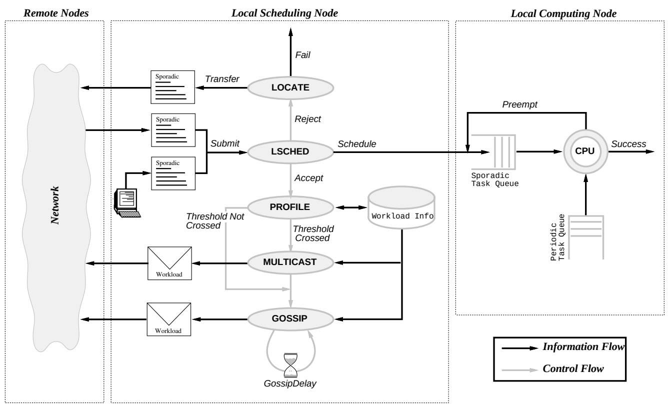

Figure 3. Protocol information/control flow

3. Performance Evaluation

Simulation Model and Metrics: We evaluated our Load Profiling Algorithm (LPA) on a system with six nodes. Each one of the nodes in the system is assigned a set of critical, periodic tasks. In addition to these critical, periodic tasks, the system is required to schedule essential, sporadic tasks, which are submitted to the individual nodes in the system. For each node in the system, the arrivals of these sporadic tasks is a Poisson process, with a mean interarrival time of λ_i. The service (execution) time for the sporadic tasks follows an exponential distribution, with a mean of μ_i. The deadline of each sporadic tasks is chosen so as to make the task laxity $(D_j - A_j - C_j)$ follow a normal distribution, with a mean of avg_i and a standard deviation of σ_i.

The baseline model for our simulations is summarized in figure 4. To model the overhead of task transfer between nodes, we introduced a task transfer delay of 5 units of time, incurred every time a task is forwarded from one node to another node. Furthermore, we introduced a communication overhead of 1 unit of time, incurred every time a message is communicated in the system.

Node	Periodic Tasks		Sporadic Tasks			
		Total Util.			Laxity	
	N	$\sum_{i=1}^{N} \frac{C_i}{P_i}$	λ_i	μ_i	avg_i	σ_i
0	3	0.700	0.01	0.02	100	50
1	2	0.417	0.01	0.02	100	50
2	2	0.500	0.01	0.02	100	50
3	3	0.783	0.01	0.02	100	50
4	2	0.350	0.01	0.02	100	50
5	3	0.250	0.01	0.02	100	50

Figure 4. Baseline task characteristics

To measure the *network-wide* load due to the arrival of sporadic tasks we define the demand ratio W. For a simulation of t time units, if I is the total number of idle cycles during that period on all the nodes—in the absence of any sporadic tasks—and S is the number of execution cycles requested by all the sporadic tasks occuring on every node during t, then the demand ratio is defined as $W = S/I$. In all the subsequent graphs, the horizontal (X) axis corresponds to the demand ratio. To measure the *efficiency of scheduling*, we use the *guarantee ratio* G. Since the periodic tasks are always guaranteed, G is defined as the total number of sporadic tasks guaranteed network-wide over the total number of sporadic tasks submitted network-wide. In all the subsequent graphs, the vertical (Y) axis corresponds to the guarantee ratio. Each data point in the following graphs is the average of enough simulation runs to guarantee a 90% confidence interval.

The middle curve in figure 5 shows the baseline simulation results. As expected, the percentage of sporadic tasks that are scheduled successfully declines as the demand ratio increases. Notably, when the demand on the system is twice as much as there are cycles to spare, the guarantee ratio drops down only to about 70%. This *"higher-than-50%"* ratio indicates that when the system is overloaded, sporadic tasks with smaller utilization requirements are preferred over others.

Effect of Task Execution Time: Figure 5 also shows the guarantee ratio for two more experiments. For the first experiment (the top curve), the mean execution time is set to 25 units of time ($\mu = 0.04$), thus making the laxity ratio equal to 4. This very large laxity ratio is the reason the algorithm achieves a high guarantee ratio, even under overloaded conditions. For the second experiment (the bottom curve), the mean execution time is set to 100 units of time ($\mu = 0.01$), thus making the laxity ratio equal to 1. This means that most tasks do not get any chances for reconsideration, once the first attempt to find a candidate target node fails. Also, the fact that the execution requirements are demanding, decreases the number of candidate target nodes. However, because of the load-profiling scheme being used, the nodes are not equally balanced, and thus the algorithm is still able to find some nodes to transfer sporadic tasks and guarantee some of them.

Effect of Task Laxity: Figure 8 shows the guarantee ratio for four experiments that were conducted to study the task laxity effect. The first experiment considers small laxities with a distribution of $N(30, 15^2)$ (*i.e.* laxity ratio = 0.6). The second experiment considers moderate laxities with a distribution of $N(60, 30^2)$ (*i.e.* laxity ratio = 1.2). The third experiment considers large laxities with a distribution of $N(100, 50^2)$ (*i.e.* laxity ratio = 2). Finally, the fourth experiment considers very large laxities with a distribution of $N(300, 100^2)$ (*i.e.* laxity ratio = 6).

Figure 8 shows that when the laxity increases the number of sporadic tasks guaranteed to meet their deadlines increases. For a moderate load of $W = 0.5$, and a laxity ratio of 0.6, the guarantee ratio is 84%, while for a laxity ratio of 6, this guarantee ratio is almost 100%. This increase in the guarantee ratio is only achievable under light or moderate loads. When the system becomes overloaded, this improvement is significantly diminished. For example, when $W = 2.0$, increasing the laxity ratio from 0.6 to 1.2, increases the guarantee ratio from 63% to 68%; increasing the laxity ratio from 1.2 to 2, increases G from 68% to 71%, while increasing the laxity ratio from 2 to 6, increases G from 71% to 73% only.

One can also see that when the system becomes excessively overloaded, increasing the task laxity does not benefit the guarantee ratio. This is also true for medium and heavy loads. After a certain threshold value, the increase in the task laxity does not result in more sporadic tasks being guaranteed.

Comparison with Other Algorithms: Figure 6 shows the results of another set of experiments under the baseline parameters. Figure 6 shows that the performance of our LPA protocol is much better than that of a protocol that utilizes a *Local Scheduling Algorithm* (LSA), and that it approaches the performance of an *Oracle Algorithm* (OA). The LSA and OA protocols can be thought of as defining lower and upper bounds on the attainable performance of our LPA protocol. Using the LSA protocol, if a sporadic task cannot be guaranteed timely execution locally, no attempts are made to forward it to a remote node. The OA protocol, on the other hand, works exactly like our algorithm, except that perfect information about node workloads is available at no overhead cost.

Figure 6 also shows the performance of two versions of our LPA protocol. These two versions differ in their reforwarding policies. The LPA protocol we considered so far allows multiple forwardings. Another possible scenario would be an LPA protocol without reforwarding; it enables the forwarding of sporadic tasks only once. Figure 6 shows that LPA with reforwarding performs better than LPA without reforwarding. This is expected since LPA with reforwarding would give *"extra chances"* for the successful scheduling of a sporadic task when inaccurate workload information is used to forward that task to a node that is incapable of granting its execution needs. However, Figure 6 shows that the difference between LPA with reforwarding and LPA without reforwarding is small, especially under moderate and heavy system loads.

The fact that LPA without reforwarding delivers most of the performance gains achievable using LPA with reforwarding could be thought of as a generalization of the Markovian analysis of Mitzenmacher [9], which considers a dynamic scheduling policy that randomly selects d out of n servers in a distributed system and then chooses one of these d servers based on some performance metric (*e.g.,* queue length). The analysis and simulations in [9] show that a d value of 2 seems to deliver most of the possible performance gains. LPA without reforwarding is a scheduling policy that examines exactly 2 servers for possibly executing an incoming sporadic task. The first server is the server to which the sporadic task is submitted, and the second server is the one that is chosen (and to which the task is forwarded) through the location policy. LPA with reforwarding could be thought of as a scheduling policy that examines d servers through successive forwarding, where $2 \ll d \leq n$. While the results in [9] were only targetted at systems that attempt to balance their load, our simulations illustrated in figure 6 suggest that these results also hold for systems that attempt to profile their load.

Figure 7 shows a baseline comparison of our LPA protocol to other *load-cognizant* algorithms, namely the *focused addressing* and *bidding* mechanisms [14], as well as to *load-incognizant* algorithms, namely a random forwarding mechanism and a no-forwarding (local scheduling only) mechanism. Our LPA protocol performs demonstrably better than all others, especially under moderate and heavy loads. For example, under a moderate-to-heavy load (*e.g.,* a demand ratio of 1), LPA offers a 20% improvement over the no-forwarding mechanism, an 18% improvement over the random forwarding mechanism, a 10% improvement over the focussed addressing mechanism, and a 5% improvement over the bidding mechanism. When the system becomes overloaded (*e.g.,* a demand ratio of 2 or more), the performance of load-cognizant techniques tend to coincide with one another. This happens because in an overloaded system load-profiling degenerates into load-balancing, since all nodes become *"equally"* overloaded.

It is interesting to note that in an overloaded system, the distinction between load-cognizant techniques and load-incognizant techniques is still manifest. For a demand ratio of 2, load-cognizant techniques seem to offer an 8% improvement in performance over load-incognizant techniques. Another interesting observation is that in a lightly-loaded system, the significance of the forwarding policy being used—whether random forwarding, focussed addressing, bidding, or load-profiling—is diminished significantly. For example, in a lightly-loaded system with a demand ration of 0.25, LPA outperforms random forwarding by only 2.5%.

4. Summary

Load Profiling—a concept that stands in sharp contrast to the traditional *load balancing* concept, often used for load management in distributed systems—is an effective technique to schedule sporadic tasks in a real-time distributed system. Our current work involves applying the load-profiling ideas presented here to other resource management problems in real-time systems.

References

[1] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[2] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 222 – 231, December 1993.

[3] M.L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, SE-15, December 1989.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12:662–675, May 1986.

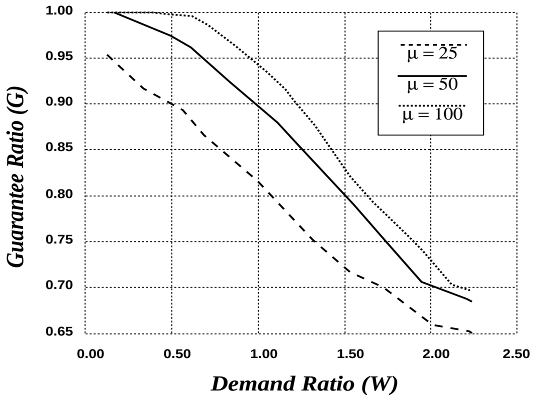

Figure 5. Effect of execution time

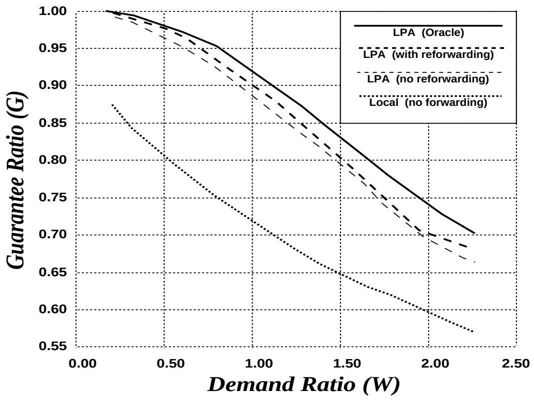

Figure 6. Performance bounds

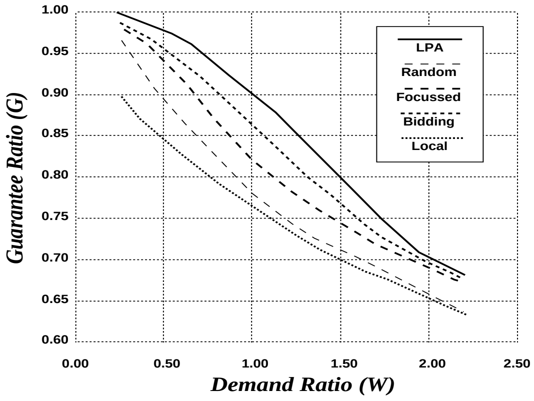

Figure 7. Comparison with other algorithms

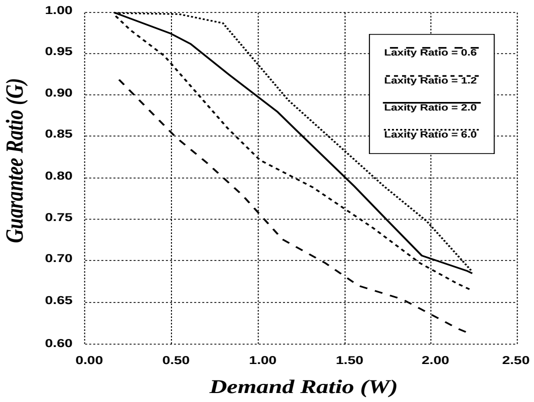

Figure 8. Effect of task laxities

[5] R. L. Graham *et al.* Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[6] M. Hailperin. *Load Balancing Using Time Series Analysis for Soft Real Time Systems with Statistically Periodic Loads.* PhD thesis, Stanford University, Computer Science Department, 1994. Also TR: CS-TR-94-1514.

[7] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the Assocation of Computing Machinery*, 20(1):46–61, January 1973.

[8] C. McGeoch and J. Tygar. When are best fit and first fit optimal? In *Proceedings of 1988 SIAM Conference of Discrete Mathematics*, 1988.

[9] Michael Mitzenmacher. *Large Markovian Particle Processes and Some Applications to Load Balancing.* PhD thesis, University of California, Berkeley, Berkeley, CA, 1996.

[10] K. Ramamritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, C-38, August 1989.

[11] Krithi Ramamritham and John Stankovic. Scheduling strategies adopted in spring: An overview. Technical Report COINS-TR-91-45, University of Massachusetts at Amherst, December 1991.

[12] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource reclaiming in real-time. In *Real-Time Systems Symposium*, pages 41 – 50, December 1989.

[13] John Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Software Engineering*, pages 1141–1152, October 1985.

[14] John Stankovic and Krithi Ramamritham. The Spring Kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, 23(3):54–71, July 1989.

[15] John Stankovic, Krithi Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, pages 1130–1143, December 1985.

[16] John Stankovic, Krithi Ramamritham, and S. Cheng. The Spring Kernel: A new paradigm for real-time systems. *IEEE Software*, pages 54–71, May 1992.

[17] S. R. Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Real-Time Systems Symposium*, pages 22 – 33, December 1994.

[18] Songnian Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems.* PhD thesis, University of California Berkeley, Computer Science Department, 1987. Also TR: CSD-87-376.