LOAD PROFILING IN DISTRIBUTED REAL-TIME SYSTEMS*

"One Size Doesn't Fit All"

Azer Bestavros
(best@cs.bu.edu)

Boston University
Computer Science Department
111 Cummington street
Boston, MA 02215

Abstract

Load balancing is often used to ensure that nodes in a distributed systems are equally loaded. In this paper, we show that for real-time systems, load balancing is not desirable. In particular, we propose a new load-profiling strategy that allows the nodes of a distributed system to be unequally loaded. Using load profiling, the system attempts to distribute the load amongst its nodes so as to maximize the chances of finding a node that would satisfy the computational needs of incoming real-time tasks. To that end, we describe and evaluate a distributed load-profiling protocol for dynamically scheduling time-constrained tasks in a loosely-coupled distributed environment. When a task is submitted to a node, the scheduling software tries to schedule the task locally so as to meet its deadline. If that is not feasible, it tries to locate another node where this could be done with a high probability of success, while attempting to maintain an overall load profile for the system. Nodes in the system inform each other about their state using a combination of multicasting and gossiping. The performance of the proposed protocol is evaluated via simulation, and is contrasted to other dynamic scheduling protocols for real-time distributed systems. Based on our findings, we argue that keeping a diverse availability profile *and using* passive bidding *(through gossiping) are both advantageous to distributed scheduling for real-time systems.*

Keywords: Distributed systems; real-time systems; scheduling; performance evaluation; load-profiling.

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

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. Most tasks in such systems have strict execution deadlines, and depending on the strictness of these deadlines, tasks are categorized as either *critical* or *essential* [13, 22]. 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*. On the other hand, if missing a task's deadline is not catastrophic, then the task's deadline is considered to be *firm* or *soft*, and the task is categorized as *essential*. The difference between firm deadlines and soft deadlines is related to the consequence of missing the deadline. If missing a deadline implies that the task must be discarded, then the dealdine is termed *firm*. Finishing the execution of a task past its firm deadline doesn't add any value to the system. 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.[1] Research in scheduling periodic tasks was ushered by the pioneering Rate-Monotonic Scheduling work of Liu and Layland [10], which was followed by several projects that aimed to catalyze an improvement in the state of the practice for real-time systems engineering based on a solid, analytical foundation for real-time resource management. Examples include the Ada RTSIA and RMARTS Projects at the Software Engineering Institute [19], which led to several results in scheduling periodic tasks with synchronization requirements [18], mode change requirements [17], specified in Ada [16]. Another leading effort in scheduling periodic tasks was the introduction of the imprecise computation paradigm by Chih, Liu and Chung [3, 21]. Using that paradigm, rather than attempting to execute each task until completion—possibly missing the task's deadline—a trade-off is made, whereby the quality of the result is traded off for timeliness.

Most of the tasks in a real-time system are likely to be essential tasks (*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*. Two common approaches for servicing soft-deadline sporadic tasks are *background processing* and *polling*. Using the background processing approach, sporadic tasks are serviced whenever the processor is idle and no periodic requests are pending. Using the polling approach, a periodic task is created to service sporadic tasks. At regular intervals, this polling task is

[1] For example, in an aircraft with a velocity of 900 km/hour, the task responsible for computing the aircraft's position should execute once every 100 msecs, in order to ensure a positional accuracy of 25 meters.

allowed to execute for a pre-determined period of time on behalf of one or more of the pending sporadic tasks (if any). While both of these techniques provide time for servicing aperiodic requests, they do not offer any *early* indication as to whether or not a submitted sporadic task will be able to meet its deadline. For many real-time applications, such early indication may be crucial. For example, in an embedded system used to control a robot arm, an early indication that the computation necessary to avoid a collision cannot be done in time would result in an emergency stop, thus ensuring a fail-safe system.

Early research on polling techniques to accomodate sporadic, essential tasks within a system with periodic, critical tasks include the Deferrable Server (DS) [9, 28] and Sporadic Server (SS) [22] algorithms, which allow the execution of sporadic tasks through a dedicated periodic server. This server is alloted a *budget* from the system resources after accounting for all the needs of critical tasks. This budget is replenished periodically. The DS and SS algorithms differ in their replenishment policies.[2] The DS and SS algorithms differ from polling in that they preserve (to some extent) the resources set-aside for sporadic tasks. Because of this feature, these algorithms are known as *bandwidth preserving* algorithms. They yield improved average response times for sporadic tasks because of their ability to provide immediate service. Other research on scheduling sporadic essential tasks in the presence of periodic critical tasks include the *resource reclaiming* [20] technique, which passively reclaims resources unused by critical tasks—when a task either executes less than its worst case computation time or when it is removed from the schedule—for the processing of essential tasks. In a similar fashion, *slack stealing* algorithms [4, 29] actively *steal* time from hard-deadline tasks to service aperiodic task requests.

Scheduling in a multiprocessor environment is an NP-hard problem [7] and requires *a priori* knowledge of task deadlines, computation times and start times [5]. 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. Current techniques for scheduling time-constrained tasks in a distributed system [8] could be thought of as extensions of traditional techniques for scheduling tasks in a distributed system. These techniques are based on a *load-shedding* approach that attempts to balance the system load amongst the different nodes therein.

A set of such heuristics for scheduling in distributed real-time systems is described in [26, 13]. These heuristics include *focused addressing* and *bidding*. 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 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 [13], 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

[2]Similar work includes the Transient Server [15], the Dynamic Priority Exchange, Total Bandwidth and Earliest Deadline Latest Servers [23].

scheme is invoked (in fact, the bidding scheme is invoked while communication with the focused node is in progress). Spring [14, 27] is an example of a multi-processor system that supports scheduling for real-time sporadic tasks.

In [30], *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**. Traditionally, the ultimate goal of load management protocols for distributed systems has been to ensure that nodes in a distributed system are equally loaded. In this paper, we show that for real-time systems, load balancing may not be desirable since it results in the available bandwidth being distributed equally amongst all nodes—in effect making every node in the system capable of offering almost the same bandwidth (*e.g.,* in terms of cycles per second) to incoming tasks. We show that for real-time systems, 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. Using load profiling, the system attempts to distribute the load amongst its nodes so as to maximize the chances of finding a node that would satisfy the computational needs of incoming real-time tasks.

The remainder of this paper is organized as follows. In section 2, we overview our distributed real-time system model: we introduce the notion of load profiling, contrast it to load balancing, and describe our decentralized scheduling and load-profiling protocols. In section 3, we present our simulation results. We conclude in section 4 with a summary and directions for future work.

2 Load Profiling for Distributed Real-Time Systems

In this section we start with a description of our basic model for a distributed real-time system. Next, we introduce the notion of load profiling and contrast it with load balancing. We follow that with a presentation of the various components of our distributed load-profiling and scheduling protocols, which are evaluated via simulation in section 3.

2.1 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 [10].

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 [20, 4, 29]. 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.

2.2 Local Scheduling of Periodic and Sporadic Tasks

For scheduling periodic tasks on a single processor, 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 [2]. 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 [2], 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 $t \leq D$, $\Omega_{\mathcal{T}}^{\text{EDS}}(0, t) \leq \Omega_{\mathcal{T}}^x(0, t)$, and $\Omega_{\mathcal{T}}^{\text{EDL}}(0, t) \geq \Omega_{\mathcal{T}}^x(0, t)$, 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.

2.3 Remote Scheduling of Sporadic Tasks

Following the terminology in [6], 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 (described in subsection 2.6) 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.

2.4 Load Profiling versus Load Balancing

In order to maximize the probability that a transfered sporadic task will meet its time constraint, each node has to gather information about the load at the other nodes in the system. Our scheme does not use this information to achieve a load-balanced system. On the contrary, it allows nodes to be unequally loaded so as to get a broad spectrum of available free cycles network-wide. We call this spectrum of available free cycles, the *availability profile* of the system. By maintaining an availability profile that resembles the expected characteristics of incoming time-constrained sporadic tasks, the likelihood of succeeding in scheduling these tasks increases. We use the term *load profiling* to describe the process through which the system availability profile is maintained.

Figure 1 illustrates the advantage of load profiling when compared to load balancing. In particular, when a sporadic task with high utilization requirements (*e.g.,* large number of CPU cycles needed per unit time until its deadline) is submitted to the system, the likelihood of successfully scheduling this task in a load-profiled system is higher than that in a load-balanced systems.

More specifically, consider a system with N identical nodes. Let $f(u)$ denote the probability density function for the utilization requirement of sporadic tasks submitted to that system. That is, $f(u)$ is the probability that the utilization requirement of a submitted sporadic task will be u, where $0 \leq u \leq 1$. Furthermore, 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* [6] 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 a candidate set $\mathcal{C}$, 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. Alternatively, $1 - L_{\mathcal{C}}(u)$ is the cumulative probability that the available utilization at a node selected at random from $\mathcal{C}$ will be larger than or equal to u, thus enough to satisfy the demand of a sporadic task requiring a utilization of u or more.

Thus, the probability that a sporadic task will be schedulable at a node selected randomly out of $\mathcal{C}$ is given by

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

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

$$P_k \quad = \quad 1 - (1 - P)^k = 1 - F(1 - W/N)^k \tag{2}$$

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 $u = x$.

The above solution corresponds to a system that schedules its load using the minimal possible number of nodes. In other words, 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:

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

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

The *perfect fit* implied in equation 3 may require that tasks already in the system be rescheduled upon the submission of a new sporadic task, or the termination of an existing sporadic task. Even if such rescheduling is tolerable, achieving a perfect fit is known to be NP-hard. For these reasons, 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 [11]. However, for a small value of N—which is likely to be the case in most distributed systems—best-fit outperforms first-fit. First-fit and best-fit heuristics work well when accurate information about the available utilization at all nodes in the system is available. This is not the case in a distributed environment, where a node's local view of global knowledge is often imprecise. In sections 2.6 and 2.5, we examine distributed load-profiling heuristics that are more appropriate for such environments.

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).

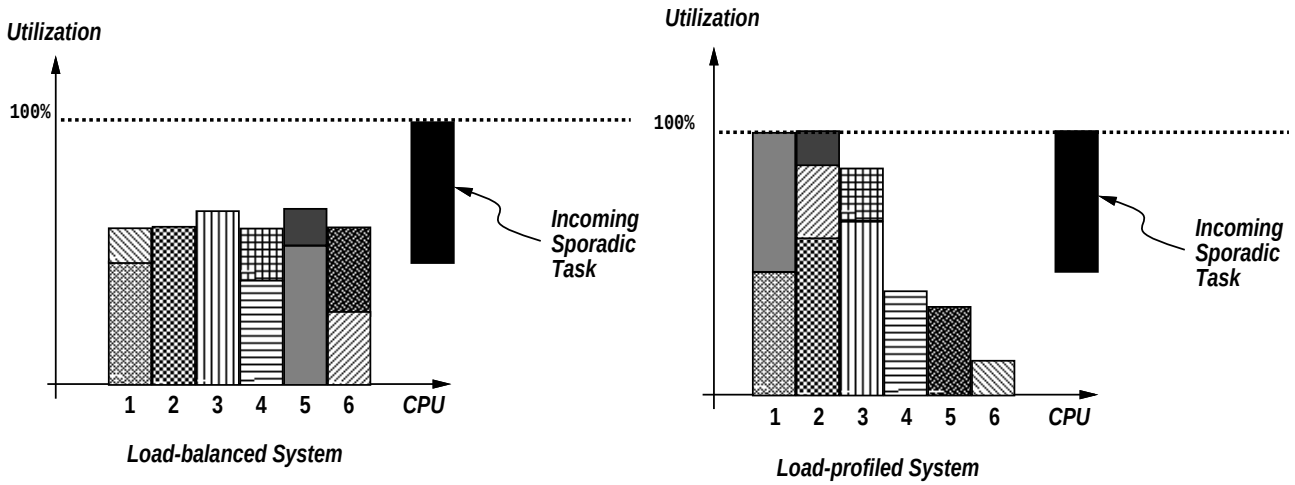

Figure 1: Load-Profiling versus Load-Balancing: An illustration

Figure 2 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 2 also shows that the advantage from using load profiling is much more pronounced when the size of the system is small.

Figure 3 shows the improvement in the schedulability of sporadic tasks when load profiling is used, under various loading conditions, and for various number of nodes in the system. This improvement is computed by dividing the percentage of sporadic tasks successfully scheduled in a load-profiled system by the percentage of sporadic tasks successfully scheduled in a load-balanced system. From figure 3 we conclude that load profiling is particularly useful when the system load is high. For example, in a distributed system with five processors, if the overall utilization of the system is 95% then it is four-times more likely that a sporadic task will be schedulable when load-profiling is used than it is when load-balancing is used.

2.5 Distributed Load-Profiling

The simulations in figures 2 and 3 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

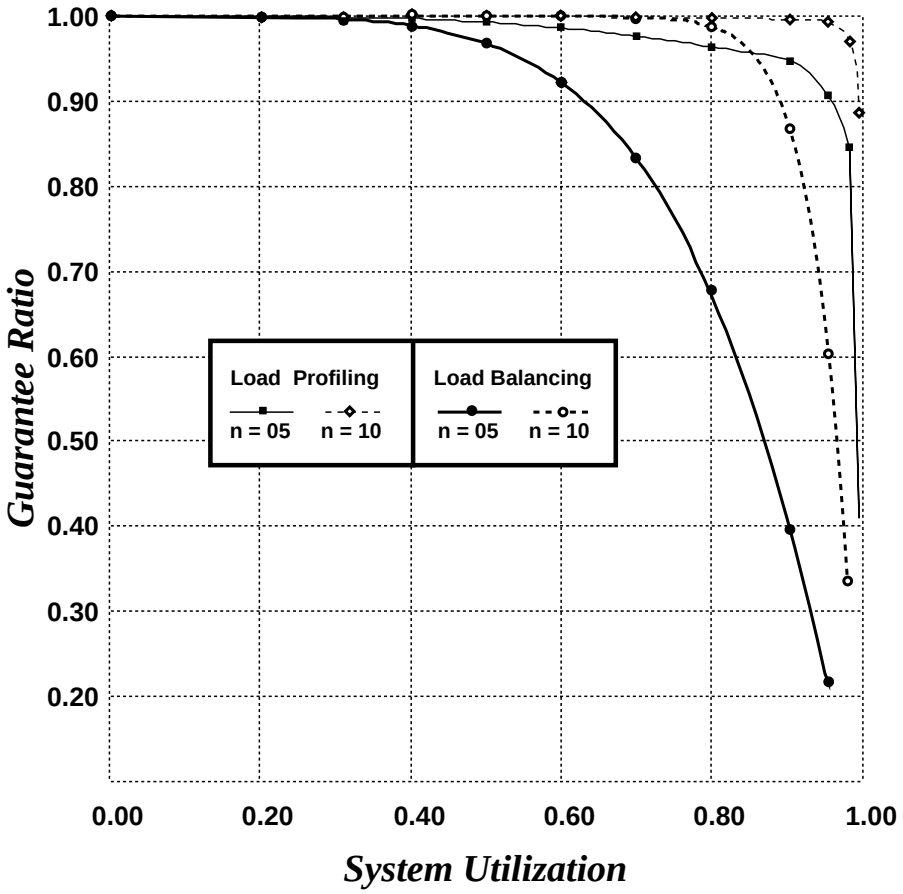

Figure 2: Load-Profiling versus Load-Balancing: Simulation Results

exchange information about their local utilization in order to enable them to construct a global (albeit approximate) view of the overall system profile.

As explained in subsection 2.2, 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. Whenever $\rho \leq \rho_l$ the node is considered to be *lightly-loaded*; whenever $\rho_l < \rho \leq \rho_m$ the node is considered to be *moderately-loaded*; whenever $\rho_m < \rho \leq \rho_h$ the node is considered to be *heavily-loaded*, but nonetheless able to schedule all the tasks assigned to it, and whenever $\rho > \rho_h$ the node is considered to be *overloaded*, and thus unable to schedule all sporadic tasks submitted to it, so it has to transfer some of its tasks to other nodes.

When the utilization of a node crosses one of these thresholds, the node sends out the information described previously in subsection 2.2, namely the localization and duration of the node's idle times

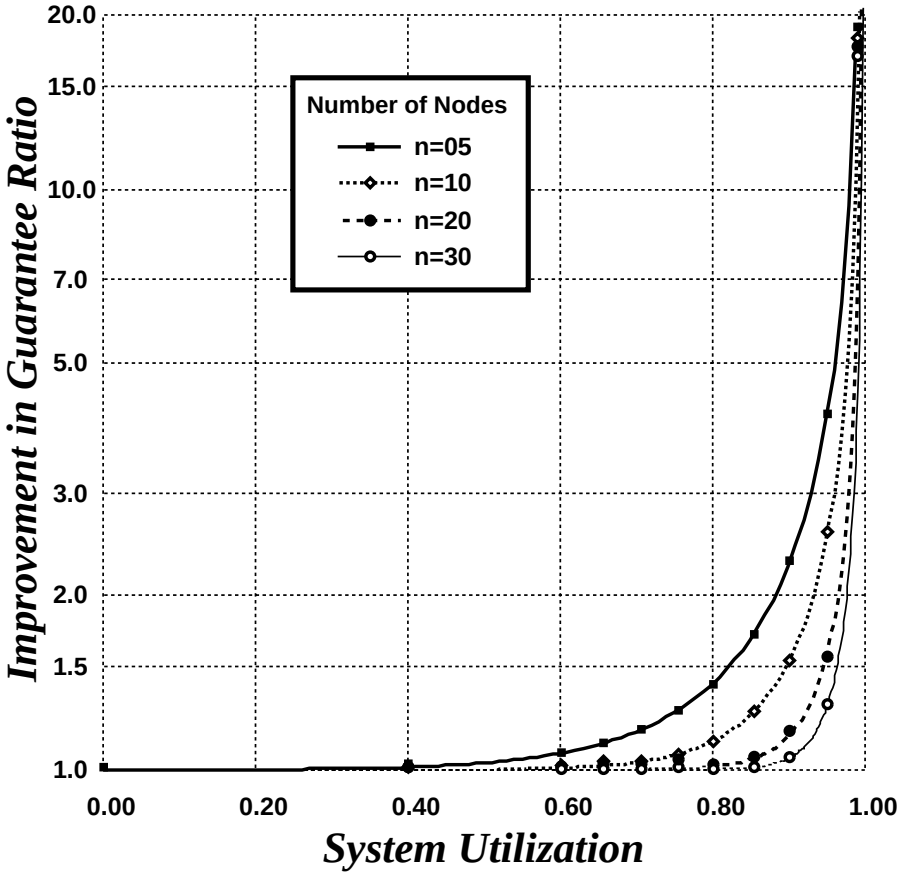

Figure 3: Load-Profiling versus Load-Balancing: Performance Improvement

and the time interval for which this information was computed. The use of threshold values allows us to distinguish bewteen a discrete number of states, each representing a significantly different load condition at the node. The exchange of local load information with other nodes in the system becomes necessary only when the node moves from one of these states to another. Obviously, a trade-off exists between the number of threshold values and the accuracy of the information exchanged in the system. On the one hand, more threshold values imply more frequent exchanges between nodes, and thus more accurate information. On the other hand, less threshold values imply less frequent exchanges between nodes, and thus less communication overhead.

When the utilization factor at a node crosses a threshold, that node communicates its load information with 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 (*i.e.* the utilization factor does not cross any of its thresholds), 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 (accordingly, it receives similar information from the neighboring nodes). A node that receives information about another node (either because of load change or gossiping) checks if the information received is newer than the one already kept. If this is

the case, it updates its information table. So, two nodes involved in gossiping can exchange up-to-date information about a third node, not directly involved in their gossip session.

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 *GossipDelay* 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 [24]. Generally speaking, by reducing the value of *GossipDelay* (*i.e.* by gossiping frequently), the size of *MulticastSet* is reduced.

2.6 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. This prediction scheme is based on the test discussed in subsection 2.2. 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 subsection 2.4, we have shown analytically that best-fit is an appropriate heuristic for choosing a target node from *CandidateSet*. However, in a distributed environment, the performance of best-fit is severely affected by the inaccuracy of the workload information communicated through the combination of gossiping and multicasting, described in subsection 2.5.[3] 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.

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 as described in subsection 2.4.

The probability of picking a node from *CandidateSet* is adjusted in such a way that the avail-

[3] As a matter of fact, it can be shown that a distributed scheduling protocol based on best-fit would perform even worse than a random scheduling protocol.

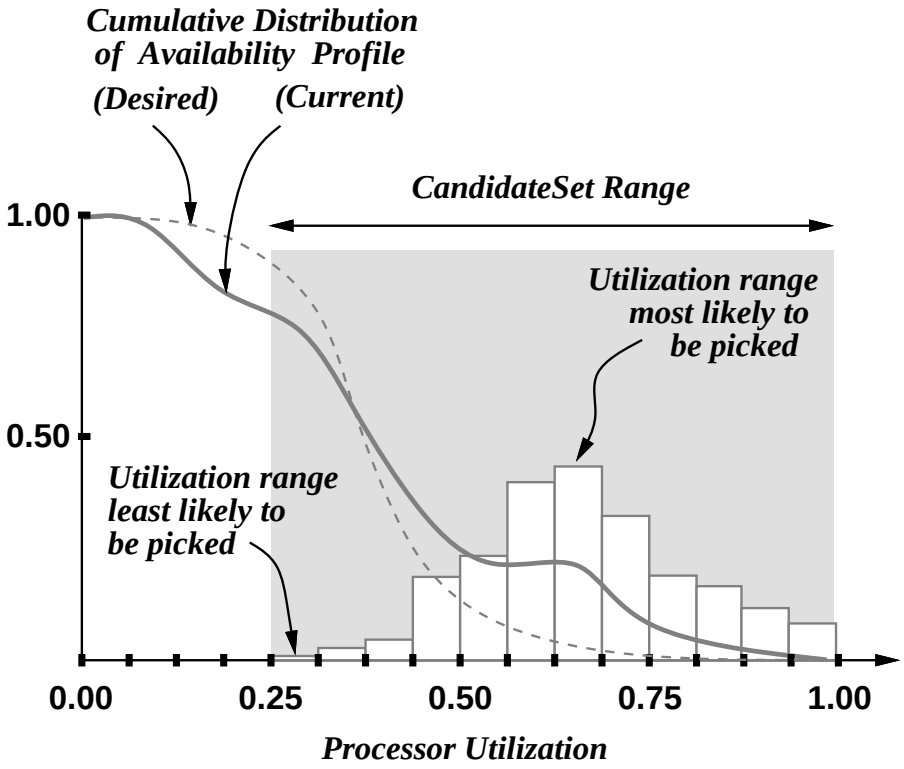

Figure 4: Maintaining a load profile that matches the characteristics of sporadic tasks

ability profile of the system is maintained as close as possible to the expected profile of incoming time-constrained sporadic tasks. Figure 4 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.

2.7 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 5.

3 Performance Evaluation

In this section, simulation results for our Load Profiling Algorithm (LPA) are presented and compared to those obtained by other dynamic scheduling schemes.

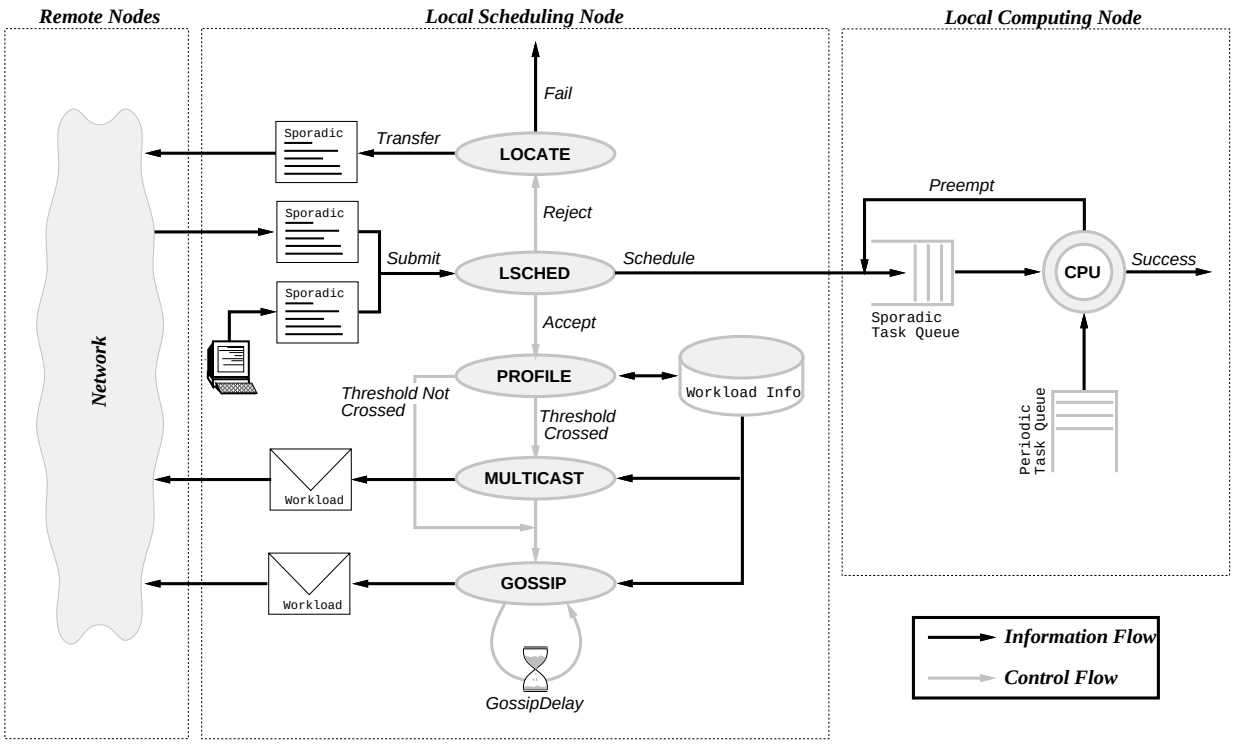

Figure 5: Information and control flow for various tasks used in our protocol.

3.1 Simulation Model and Metrics

We evaluated our LPA protocol on a system with six nodes as shown in figure 6. 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.

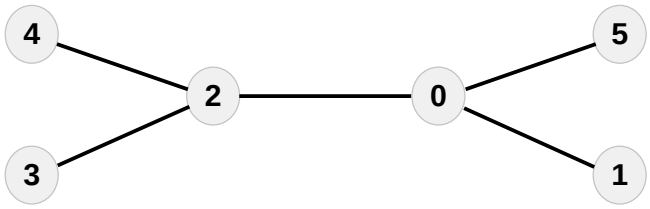

Figure 6: Network topology for the simulation model

The baseline model for our simulations—describing the characteristics of the load at each one of the six nodes—is summarized in figure 7. Notice that the parameters for sporadic tasks on all nodes

are identical. In addition to the parameters in figure 7, and in order to model the overhead of task transfer between nodes, we introduced a task transfer delay of 5 units of time per hop. This delay is incurred every time a task is forwarded from one node to another node. Furthermore, we introduced a communication overhead of 1 unit of time. This delay is incured every time a message (*e.g.,* as a result of `GOSSIP` or `MULTICAST`) is communicated in the system.

Node Id	Critical Periodic Tasks		Essential Sporadic Tasks			
	Number N	Total Util. $\sum_{i=1}^{N} \frac{C_i}{P_i}$	Arrival Rate λ_i	Exec Time (C_j) μ_i	Laxity ($D_j - A_j - C_j$) 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 7: Characteristics of periodic tasks on each node

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 $W = S/I$. Notice that this measure does not take into consideration the pattern of the arrival times of the sporadic tasks, nor does it reflect the nodes to which these sporadic tasks are submitted. So, even if W is less than or equal to 1.0, this does not mean that the system should be able to guarantee all the sporadic tasks that arrive, because of bursty arrivals that might have occured. In all the subsequent graphs, the horizontal (X) axis corresponds to the demand ratio.

To measure the performance of the algorithm, 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.[4].

The simulation results for the baseline parameters of figure 7 are shown in figure 8. 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.

[4]Most of the time five experiments were sufficient to achieve that level of confidence

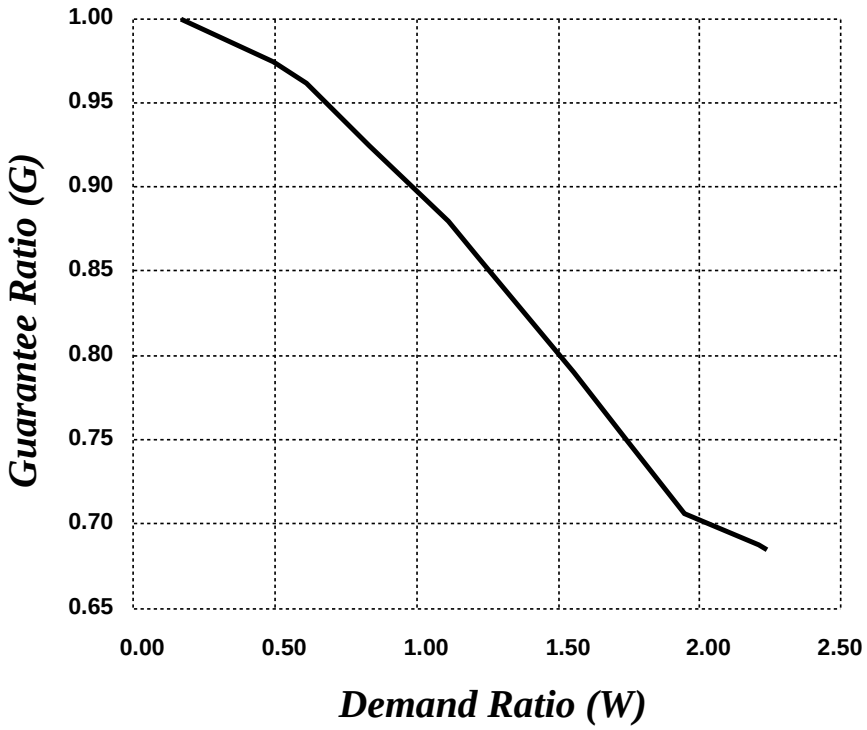

Figure 8: Siumlation results showing the guarantee ratio for the baseline parameters.

3.2 Effect of Task Execution Time

Figure 9 shows the guarantee ratio for three experiments. With the exception of the mean execution time for sporadic tasks, the parameters used in these experiments are identical to the baseline parameters of figure 7.

For the first experiment, 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 mean execution time is set to 50 units of time ($\mu = 0.02$), thus making the laxity ratio equal to 2. Due to the larger execution times of the tasks, the guarantee ratio is not so high as in the previous case. The situation gets even worse in the third experiment, when 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.

3.3 Effect of Task Laxity

Figure 10 shows the guarantee ratio for four experiments. With the exception of the laxity of sporadic tasks, the parameters used in these experiments are identical to the baseline parameters of figure 7.

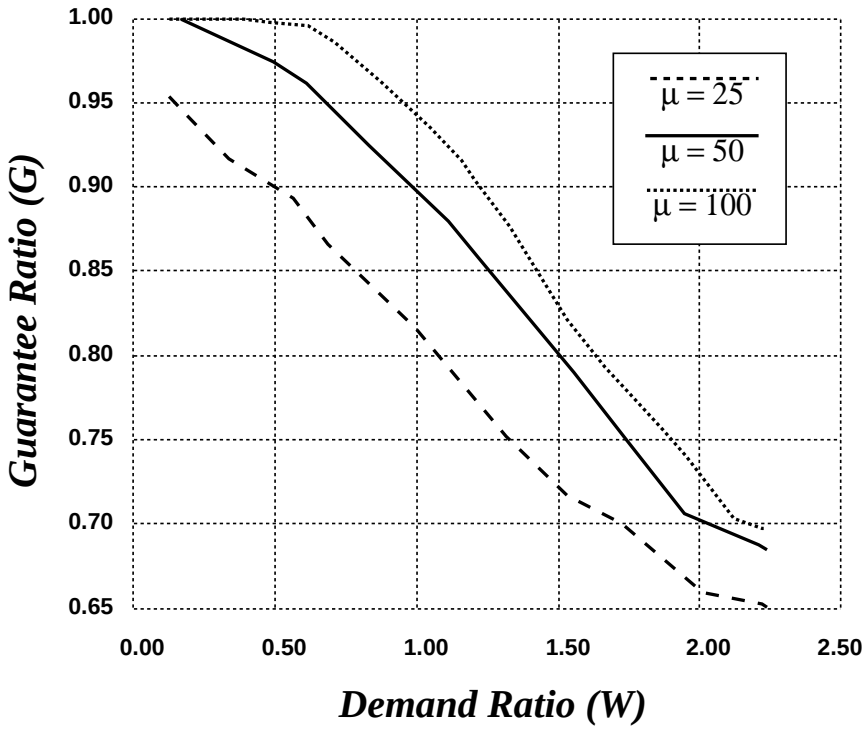

Figure 9: Effect of execution time on guarantee ratio.

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 10 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. Figure 11 shows that for $\mu = 0.02$, increasing the task laxity from $N(300, 100^2)$ to $N(450, 150^2)$, and from $N(450, 150^2)$ to $N(600, 200^2)$ does not increase the number of sporadic tasks guaranteed. The threshold value for the task laxities in this specific case is 300, thus a laxity ratio of 6.

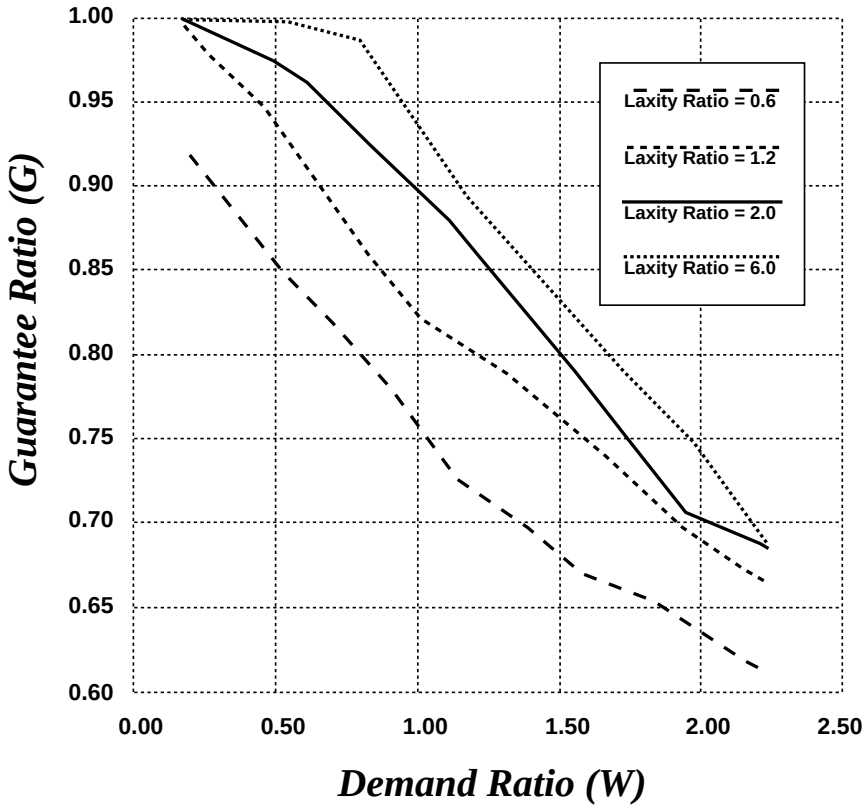

Figure 10: Effect of laxities on guarantee ratio.

3.4 Comparison with Other Dynamic Algorithms

Figure 12 shows the results of another set of experiments under the baseline parameters shown in figure 7. Figure 12 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 12 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—it enables multiple forwarding of sporadic tasks from one node to another. Another possible scenario would be a LPA protocol without reforwarding; it enables the forwarding of sporadic tasks only once. Figure 12 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 12 shows that the difference between LPA with reforwarding and LPA without reforwarding is small, especially under moderate

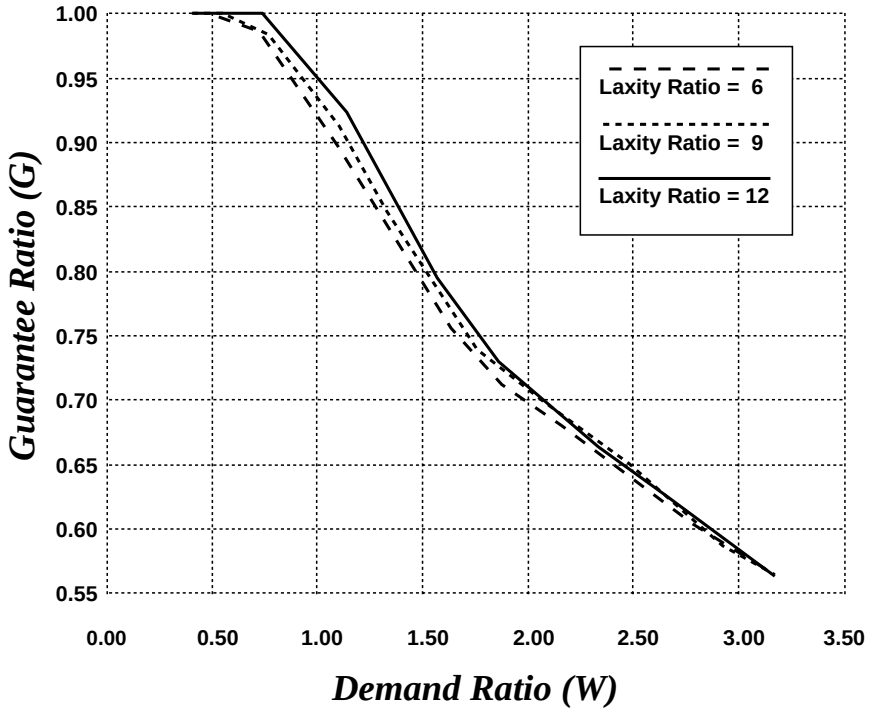

Figure 11: Effect of very large laxities on guarantee ratio.

and heavy system loads. Such a small improvement in performance may not be enough to warrant the use of reforwarding, especially when the overhead of forwarding is taken into consideration.

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 [12], 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 [12] 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 [12] were only targetted at systems that attempt to balance their load, our simulations illustrated in figure 12 suggest that these results also hold for systems that attempt to profile their load.

Figure 13 shows a comparison of our load-profiling algorithm to other *load-cognizant* algorithms, namely the *focused addressing* mechanism and the *bidding* mechanism [25], as well as to *load-incognizant* algorithms, namely a random forwarding mechanism and a no-forwarding (local scheduling only) mechanism. The parameters used for these experiments are those of the baseline prameters shown in figure 7. 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%

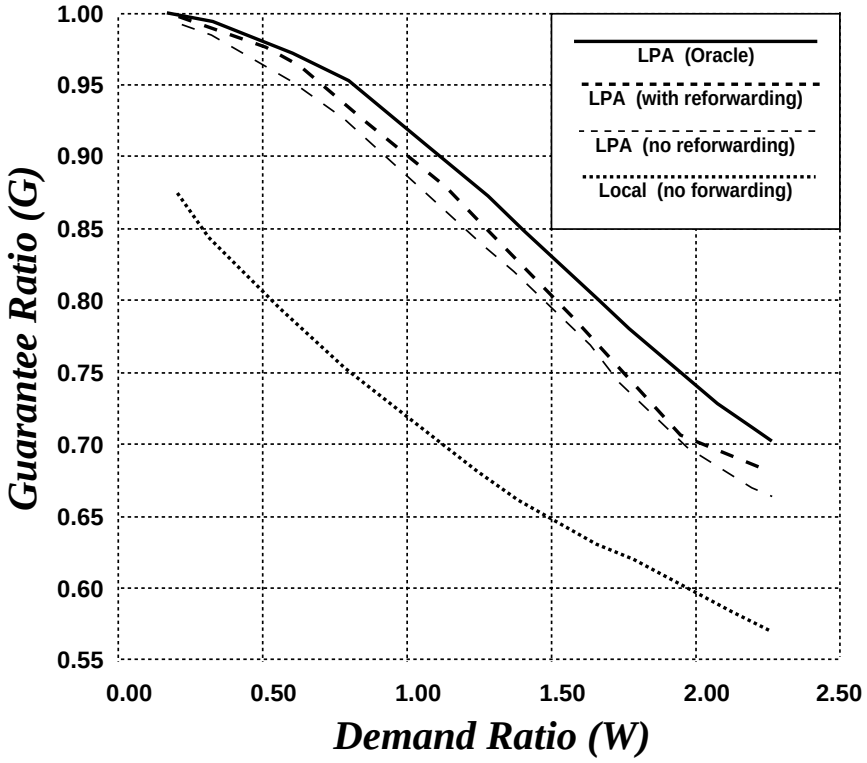

Figure 12: Lower and upper performance bounds.

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 notice that in an overloaded system, the distinction between load-cognizant techniques and load-incognizant techniques is still manifest. In particular, 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 Conclusion

Dynamic scheduling of tasks with execution deadlines in a distributed time-critical environments is known to be an NP-hard problem. In this paper, a heuristic protocol was presented and evaluated. Our approach is a dynamic one that tries to maximize the number of sporadic tasks that are accepted

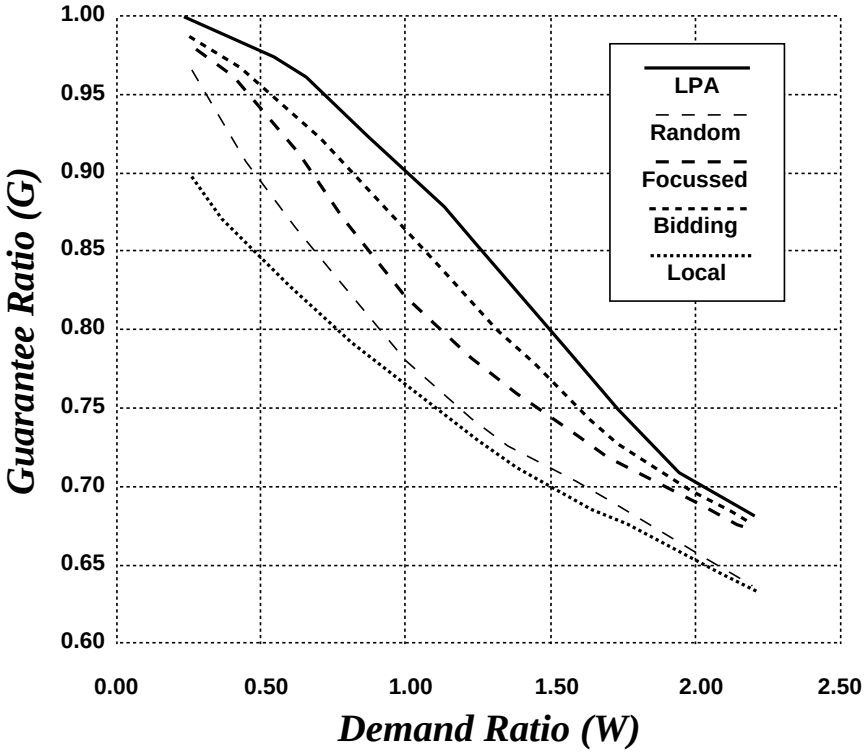

Figure 13: Comparison with other scheduling algorithms.

for execution—and thus guaranteed to finish before their deadlines—in the presence of critical periodic tasks that must always meet their deadlines.

The main concept introduced and evaluated in this paper is that of *load profiling*—a concept that stands in sharp contrast to the traditional *load balancing* concept, often used for load management in distributed systems. Using load profiling, a distributed system attempts to maintain an availability profile that matches the expected workload distribution. The simulation results we have presented confirm the superiority of our load-profiling-based dynamic scheduling protocols for distributed systems, when compared to previous load-balancing-based protocols. This improved performance is more pronounced in moderately-loaded and heavily-loaded systems than it is in lightly-loaded or overloaded systems.

Our current research work involves extending the ideas presented in this paper to allow for value-cognizant admission control protocols based on load profiling in a distributed environment.

Acknowledgments

I would like to thank Dimitri Spartiotis for his work on implementing parts of the simulator for the distributed load-profiling algorithm, and Krithi Ramamritham and Jack Stankovic for their valuable remarks on an earlier version of this work [1].

References

[1] Azer Bestavros and Dimitrios Spartiotis. Probabilistic Job Scheduling for Distributed Real-time Applications. In *Proceedings of the First IEEE Workshop on Real-Time Applications*, New York, NY, May 1993.

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

[3] Jen-Yao Chung, Jane Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transaction on Computers*, 19(9):1156–1173, September 1990.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the 8th IEEE Real-time Systems Symposium*, pages 261–270, December 1987.

[10] 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.

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

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

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

[14] 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.

[15] S. Ramos-Thuel and J. K. Strosnider. The transient server approach to scheduling time-critical recovery operations. In *Real-Time Systems Symposium*, pages 286 – 295, December 1991.

[16] L. Sha and J. Goodenough. Real-time scheduling theory and ADA. *IEEE Computer*, April 1990.

[17] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1, 1989.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 9 1990.

[19] Lui Sha, R. Rajkumar, Sang Son, and Chun-Hyon Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

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

[21] Wei-Kuan Shih, Jane Liu, and Jen-Yao Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM journal of Computing*, July 1991.

[22] B. Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[23] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 2 – 11, December 1994.

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

[25] 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.

[26] 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.

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

[28] Jay Strosnider. *Highly Responsive real-time token rings.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1988.

[29] 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.

[30] 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.