Deadline Assignment in a Distributed Soft Real-Time System

Ben Kao* Hector Garcia-Molina†

April 16, 1993

Abstract

In a distributed environment, tasks often have processing demands on multiple different sites. A distributed task is usually divided up into several subtasks, each one to be executed at some site in order. In a real-time system, an overall deadline is usually specified by an application designer indicating when a distributed task is to be finished. However, the problem of how a global deadline is automatically translated to the deadline of each individual subtask has not been well studied. This paper examines (through simulations) four strategies for subtask deadline assignment in a distributed soft real-time environment.

Keywords: soft real-time, distributed systems, deadline assignment, scheduling.

1 Introduction

Consider a radar surveillance system whose task is to track flying objects, decide whether an object is friendly or hostile, and in the latter case, come up with a combat strategy. This system requires the cooperation of several different components: a radar sensor component gathering radar images, a vision system extracting the features of objects, a database component matching those features with those of known objects, an expert system selecting the best defense plan, and so on. Each component may run on a separate device or computer, using its own scheduling policy. For instance, the database lookup may be done on a backend database server that uses an earliest (local) deadline first scheduler.

*Princeton University Department of Computer Science. Current address: Department of Computer Science, Stanford University, Stanford, CA 94305. e-mail: kao@cs.stanford.edu

†Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

In this example, several components of the system are involved between the first sighting of an object and a defensive action like launching a missile. The system specification indicates how much time the system as a whole is allowed to respond to the threat of an intruder. In other words, a deadline of the complete task (from sighting to action) is specified. However, the system is faced with the problem of assigning deadlines or priorities to each of the subtasks. For example, the subtask that is doing a database lookup for a particular object will be running on the database server, competing against other database tasks. What local deadline is to be assigned to the lookup subtask in this case? Answering this question is the main goal of this paper.

Our focus is on *soft real-time* systems. In such systems, it is very difficult to guarantee that all deadlines will be met, and hence one tries to *minimize* the number of deadlines that are missed. Guaranteeing all deadlines may be hard because it is impossible to impractical to place an upper bound on the load. In our radar example, for instance, one may guess the maximum number of objects that may suddenly appear, but there is no way of knowing for sure that this maximum will never be exceeded. Another reason why soft real-time systems may miss deadlines is because the tasks they run are complex and it is hard to predict exactly how long they will run or what resources they may need. For instance, it is hard to know in advance how many rules the expert system in our example will trigger. In the database component, the running time of a search will depend on what other transactions are concurrently running and holding locks. Furthermore, disk access times will depend on where the previous request left the disk arm, again hard to determine in advance.

In this paper we study the *subtask deadline assignment problem*, or SDA for short. A given task is to be executed and completed by a specified deadline. The task executes several subtasks, each at possibly different system components. As each subtask is submitted to its component, a local deadline must be assigned to it. At each component, there may also be local tasks that just run at that component and that compete for resources with the subtasks assigned there.

To study the SDA problem, we make three assumptions:

- The subtask deadlines are to be assigned *on-line*, as opposed to a-priori when the global task is defined or first submitted. On-line assignment is superior in soft real-time systems, where the types of subtasks or their durations may be unknown in advance. We assume a subtask receives its deadline just before it is submitted for execution at its corresponding component.

- The scheduler at each component is independent of the others. There is no global scheduler that instructs each component scheduler what to do. Each scheduler makes decisions based solely on the subtasks (and their deadlines) that have been presented to it for execution, without consulting other schedulers. We believe that large systems are built out of preexisting components. Each component will have its own scheduling policy and will be unable or unwilling to coordinate or subordinate its scheduling decisions with (or to) others.

- Each system component is unique. If a subtask must be executed at a particular component, it must run there. There is no load balancing, i.e., an overloaded component cannot ship subtasks to other components.

To illustrate how the choice of a SDA strategy affects a real-time system, let us consider the following example. Suppose we have a real-time task T that consists of two subtasks T_1 and T_2 to be executed in series at components C_1 and C_2 respectively. Before a subtask is submitted for execution, a deadline has to be assigned to it so that the component real-time scheduler knows about the relative priority of the subtask compared to other tasks. Since T_2 is the last subtask of T, its deadline should just be the (ultimate) deadline of T. If T_1 also adopts the same ultimate deadline, there will be a problem when the system load is high: In a tight situation, when tasks are being finished close to their deadlines, this scheme would leave little or no slack for the execution of T_2. Task T would thus have a high probability of missing its deadline. This discussion suggests that an earlier deadline should be assigned to T_1 instead. The question is: how early? If the deadline is not early enough, we have the same problem; on the other hand, if the deadline is too early, then the C_1 scheduler will be fooled regarding the urgency of subtask T_1. Other "more urgent" tasks will have to give way to this "false urgent" subtask and priority inversion results[1].

We believe that the SDA problem is a central one in distributed soft real-time systems. Notice that it even encompasses the communication subsystem. That is, sending messages can be viewed as another type of subtask. For instance, let us return to our two task example: $T = T_1, T_2$. Say that after T_1 completes, it is necessary to send a message from C_1 to C_2 containing the inputs for T_2. Then after T_2 finishes it is necessary to ship the final result to some other site. The two transmissions can be seen as two additional subtasks, T_a, T_b, so that the global task is really $T = T_1, T_a, T_2, T_b$. The communications subsystem needs to be given

[1]Priority inversion occurs when a high priority task (or a more urgent task in our context) is blocked by a lower priority one.

a deadline when T_a (and T_b) is submitted, so again we see a need for SDA. Also note that the SDA problem may arise *within* a single site that contains multiple subsystems such as the CPU, the disk, and memory. For example, even within a single database system, sub-deadlines may have to be used when requesting a disk block read, a buffer page allocation, or a processor for computation.

As we proceed, we will address the following questions:

1. What are the options in choosing a SDA strategy?
2. How does a given SDA strategy affect a task's chances of meeting its deadline?
3. What are the underlying reasons for the observed effects of a SDA strategy?
4. Given a set of system parameters (e.g., load, slack of tasks) and performance goals (e.g., to minimize the number of tardy tasks, to treat tasks fairly), how would one choose a SDA strategy?
5. For those strategies that use a prediction of task execution times, how accurate must the prediction be to maintain a low level of missed deadlines?

The rest of this paper is organized as follows. In Section 2, we mention some related work. Section 3 describes the logical base model for our study. Different SDA strategies are introduced in Section 4. A brief description of our simulation experiments is contained in Section 5. In Section 6 we display and analyze the results of our experiments, while Section 7 discusses other variations of our base model. In Section 8 we present conclusions.

2 Related Work

A lot of research has been done on real-time scheduling in various environments, be it I/O scheduling, processor scheduling, or transaction scheduling [3, 1, 2, 8]. However, most of this work either:

- Focuses on the characteristics of a specific system component and evaluates the performance of various scheduling policies when applied to that component. For example, there have been studies on real-time database systems [6], or communication systems [7, 12].

These studies make the tacit assumption that the deadlines of tasks when they arrive at the component are known. They do not concern themselves on whether these tasks are really subtasks of a global task that involves multiple components.

- Assumes a global scheduling strategy that controls in some way all component schedulers. For example, in [11], schedulers interact with each other to decide the best place to execute a subtask. As we have stated, here we look instead at an environment where local schedulers are autonomous and subtasks cannot move.

There are however at least two studies that are closely related to our approach and the SDA problem. Bettati and Liu [4, 5] discussed the problem of scheduling subtasks in a *hard* real-time distributed environment. Their work focuses on those systems for which global tasks can be characterized as "flow shops." In their model, global tasks consist of the same set of subtasks to be executed on nodes in the same order. The goal is to devise efficient *off-line* algorithms (assuming all the information about global tasks is known ahead) for computing a schedule of the subtasks such that all deadlines are met (if such a schedule exists). In their work, different variations of the model are studied, based on different assumptions on subtask execution time (e.g., whether all subtasks have the same execution time). Other variations, like periodic tasks, are also discussed.

Another interesting paper [10] by Pang, Livny, and Carey investigates the problem of "bias" against longer transactions under "earliest-deadline-based" scheduling policies in *real-time database systems*. The study shows that long transactions miss more deadlines compared to short ones. This is not because of tighter timing constraints (the phenomenon occurs even when long transactions have larger amount of slack), but because of their bigger size, which causes them to have "further-in-the-future" deadlines. Long transactions thus compete unfavorably with short transactions in accessing system resources.

Their approach to the problem is to assign *virtual deadlines* to all transactions. A transaction with an earlier virtual deadline is served before one with a later virtual deadline. The virtual deadline of a transaction is adjusted dynamically as the transaction progresses, and is computed as a function related to the size of the transaction. In some sense, their approach is to introduce another bias factor counter-attacking the intrinsic bias behaviour of earliest-deadline-based scheduling policies. By monitoring the system and gathering runtime statistics, a parameter in computing virtual deadlines is carefully adjusted in such a way that the linear correlation between miss ratio and transaction size is minimized.

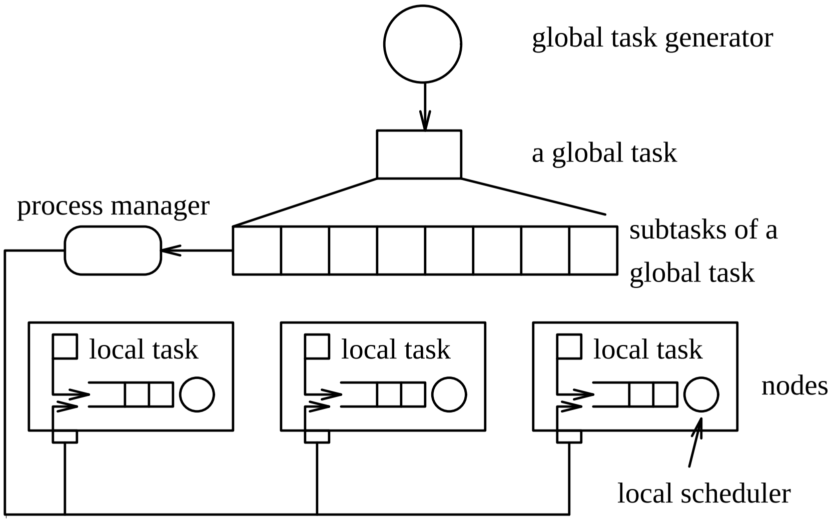

Figure 1: Model.

Our work is similar to [10] in that we both try to assign earlier deadlines to transactions (or tasks). However, in their case there is a single scheduler for a single database system. For our problem, there are multiple "resources" handled by independent schedulers. Furthermore, distributed tasks have natural breaks (subtasks) that make the assignment of deadline more natural.

3 Base Model

To study the SDA problem we postulate a simple model of the system and tasks. A distributed real-time system consists of several *nodes* representing system components (see Figure 1). Each node manages one or more resources, for example, a database, a cycle server, or a communication channel. At each node, there is a real-time scheduler prioritizing tasks according to some real-time queueing discipline, e.g., earliest deadline first.

We consider two categories of tasks: *local* and *global*. A task that visits only one node and is submitted to a scheduler only once is termed a local task. On the other hand, a global task is a *set* of several related *subtasks* (or *stages*) to be executed in series.[2] The deadline of a global task is the time by which the last subtask must complete. Each subtask is associated with an *execution node* to which it is submitted for execution.

[2] Subtasks could also be executed in parallel, but this is not considered here due to space limitations.

We further assume that a global task is controlled and monitored by a process manager. The manager runs at any node, and may be different for each global task. When a global task is generated, its process manager will submit the first subtask to the corresponding execution node. When a subtask, X, finishes, its manager is notified and immediately submits the next subtask (if there is any) of X's global task. We assume that the process manager of a global task does not consume any resources.

Associated with each task X (whether it is local, global, or a subtask) are five attributes denoted by the following functions:

$$\begin{aligned}
ar(X) &= \text{arrival (or submission) time of } X, \\
sl(X) &= \text{slack of } X, \\
dl(X) &= \text{deadline of } X, \\
ex(X) &= \textit{real} \text{ execution time of } X, \\
pex(X) &= \textit{predicted} \text{ execution time of } X.
\end{aligned}$$

The first four attributes are related by the following equation:

$$dl(X) = ar(X) + ex(X) + sl(X).$$

When a global task X is first processed by its manager, we assume its deadline $dl(X)$ is known. Since its arrival time is also known, the slack can be computed using the above equation. The execution time $ex(X)$ is *not* known in advance; we only define it here to help us in the discussion.

In some scenarios, an estimate of the execution time, $pex(X)$, may be available and can be used by the manager for making subtask deadline assignments. Furthermore, in some cases, the manager may know an estimate or prediction on the number of subtasks involved in X, plus an estimate of their duration. Most of the subtask deadline assignment policies to be discussed in the next section will utilize this information.

We also define flexibility (denoted by $fl(.)$) of a task X to be the ratio of the amount of X's slack to the execution time of X. That is,

$$fl(X) = sl(X)/ex(X).$$

Intuitively, the more flexible a task is (higher $fl(.)$), the less stringent is its timing constraint.

An additional attribute for a subtask is its *stage*, which is defined by:

$$stage(X) = i \text{ if } X \text{ is the } i\text{th subtask of its global task.}$$

We say that a subtask X is an *earlier-stage* compared to another subtask Y if they are of the same global task and $stage(X) < stage(Y)$.

Finally, there is the issue of tardy tasks, or overload management policy. Say a task X has already missed its deadline, but has not completed execution. One option is to abort X as soon as it misses its deadline, under the assumption that whatever it was doing is now useless. (Example: after analyzing the current state of the stock market, a decision is made to sell certain stock. A task X is issued to sell by a given time. If the time is exceeded, it is best to abort X, as the market conditions may have changed.) A second option is to continue to process X, under the assumption "better late than never." (Example: at a bank, customers are "guaranteed" a two second response time. However, if the guarantee is not met, it is still desirable to complete the task.) Due to space limitations, in this paper, we focus on the no abortion case (no specific action is taken when a deadline expires). However, as is briefly mentioned in Section 7, we have studied the abortion case and the results do not significantly change the relative performance of the SDA strategies.

4 SDA strategies

In this section, we propose four SDA strategies and give their formal definition. We consider a global task T that consists of m subtasks T_1, ..., T_m to be executed in series. An SDA strategy is one that determines the values of $dl(T_i)$ $(1 \leq i \leq m)$ at the time when T_i is submitted.

We demonstrate the various SDA strategies by an example shown in Figure 2. In the example, the global task T has 4 subtasks T_i, T_{i+1}, T_{i+2}, T_{i+3} remaining. These subtasks have execution times of 3, 1, 1, and 1 units respectively. Moreover, T_i arrives at time 0 (i.e., T_{i-1} finished at time 0) and the ultimate deadline of T is time 12. The amount of slack remaining for T is 6 units.

Without any knowledge on the execution times of the subtasks, the only available measure of their timing requirement is the deadline of their global task T. A simple SDA strategy would be to set the deadline of a subtask to be equal to the deadline of its global task (12 in Figure 2). We call this strategy *Ultimate Deadline* (UD).

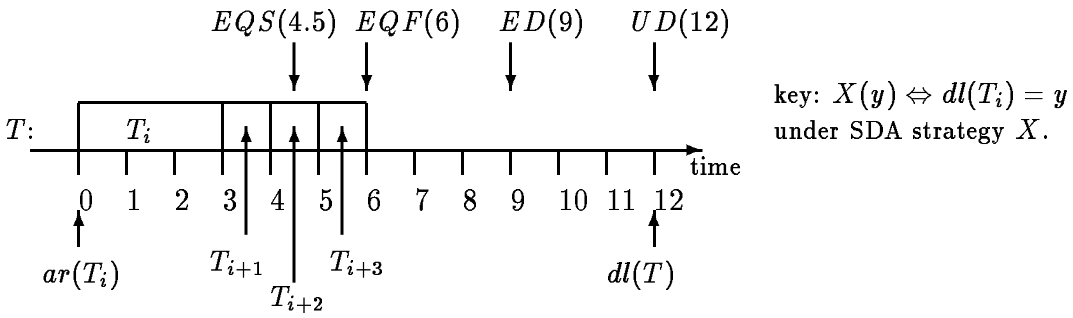

Figure 2: Example on the various SDA strategies.

(1) Ultimate Deadline (UD):

$$dl(T_i) = dl(T).$$

A problem with UD is that the time for the execution of a later-stage subtask (e.g. T_{i+1} in Figure 2) is considered slack to an earlier stage (T_i). This gives the schedulers incorrect information about how much time a subtask can be delayed in its execution without causing a missed deadline.

If an estimate of the subtask execution time is available, we can compute the effective deadline of a subtask. The effective deadline of a subtask T_i is equal to the deadline of its global task T (time 12 in the example) minus the total expected execution time of the subtasks of T following T_i (3 units)[3]. Formally, we have,

(2) Effective Deadline (ED):

$$dl(T_i) = dl(T) - \sum_{j=i+1}^{m} pex(T_j).$$

A problem common to both UD and ED is that all the remaining slack of T (6 units in the example) is allocated to the currently active subtask (T_i). This subtask thus has a low priority compared to other tasks of the system. A big portion of this slack may be consumed while the subtask is waiting for its turn in the scheduler queue. Subtasks that represent early stages of global tasks thus consume most of the slack of their global tasks. This leaves little slack for the subtasks to follow. Global tasks may therefore have a low probability of meeting their deadlines.

[3]We assume the estimates are perfect in our example.

The problem of "little slack for final-stage subtasks" gets worse when there are many local tasks in the system. This is because local tasks have only one, probably short stage. If local tasks have similar flexibility ($sl(.)/ex(.)$) as compared to global tasks, then on average they have smaller total amount of slack than global ones do. When competing for system resources, early-stage subtasks of global tasks will be discriminated against (i.e., scheduled later than other tasks), because of their much larger slack. The scheduler is therefore biased in favor of local tasks at the expense of global ones.

To avoid discrimination and to allow enough slack for final-stage subtasks, each subtask should have its fair share of its global task's slack. One way of doing it is to divide the total remaining slack (6 units in Figure 2) equally among the *remaining* subtasks (4 of them). This gives us the *Equal Slack (EQS)* strategy:

(3) Equal Slack (EQS):

$$dl(T_i) = ar(T_i) + pex(T_i) + [dl(T) - ar(T_i) - \sum_{j=i}^{m} pex(T_j)]/(m - i + 1).$$

The term in brackets represents the remaining slack. In our example, EQS would assign $dl(T_i)$ as $0 + 3 + 6/4$ or 4.5. This is shown at the top of Figure 2.

A fourth strategy is to divide the total remaining slack among the subtasks in proportion to their execution times. In this way, subtasks of the same global task do not have equal slack, but equal flexibility. We call this strategy *Equal Flexibility (EQF)*.

(4) Equal Flexibility (EQF):

$$dl(T_i) = ar(T_i) + pex(T_i) + [(dl(T) - ar(T_i) - \sum_{j=i}^{m} pex(T_j))] * [pex(T_i)/\sum_{j=i}^{m} pex(T_j)].$$

For our example, under EQF, $dl(T_i)$ is $0 + 3 + 6 * (3/6)$ or 6.

5 Simulation Model

In order to study and contrast system behavior under these SDA strategies, we developed a simulation model and performed extensive experiments. In this section we describe the model; our results are presented in Section 6.

In practice, the soft real-time systems we are trying to study can be very complex. For instance, each node may have a different scheduler type, and different execution characteristics. For example, the communications subsystem may schedule outgoing messages using a simple earliest deadline first strategy, and its subtasks may take on the order of microseconds to execute. A database server node, on the other hand, may take into account lock contention in its scheduling decisions and subtasks may take on the order of seconds. Similarly, global tasks can be extremely varied, ranging from ones with a few short subtasks, to ones lasting days and spanning many nodes.

If we model all this diversity and complexity, our results on SDA would be obscured by many irrelevant parameters. Instead, we use a very simple model that captures the essential features that impact SDA. To pick a particular example, in a given experiment we assume that all global tasks have a fixed number of subtasks, m. Clearly this is not "realistic;" as we have stated, tasks will have a variety of subtasks. If we were trying to predict performance of a particular distributed application, this assumption would not be acceptable. But if we are trying to understand how the number of subtasks affects the choice of SDA algorithm, this is the best assumption. By running different experiments with different values of m, we may be able to see that for short global transactions maybe algorithm X is better than Y. For long tasks, the reverse may be true. The same argument can be made for all the modeling assumptions we are about to make. Our point is that only by selecting a small number of simple, key parameters will we be able to understand the fundamental interactions.

Our simulator is written in the simulation language **DeNet**[9]. Each simulation experiment (generating one data point) consists of two simulation runs, each lasting one million time units (at least 100,000 tasks are generated per run, many more for high load experiments). The 95% confidence interval is $\pm$ 0.35 percentage point (much smaller for high load experiments) for the missed deadlines figures shown in later sections.

The structure of our simulation model follows the conceptual model described in Section 3 with the following characteristics.

Nodes

There are k (homogeneous) nodes in the system. Each node services their tasks according to some real-time scheduling algorithm with *no preemption*. In this paper, we use *earliest-deadline-first*[4] as the scheduling algorithm for most of our experiments, mainly due to its popularity. We

[4]Tasks in a scheduler queue are ordered in increasing deadlines; The task with the earliest deadline is served

will also discuss briefly the case when *minimum-laxity-first*[5] is used as the scheduler algorithm in Section 7.

Local Tasks

Local tasks are being generated *at each node* according to a Poisson distribution with mean interarrival time $1/\lambda_{local}$ time units. Since there are k nodes, the total average arrival rate is $k\lambda_{local}$ per unit time. Execution times of local tasks are exponentially distributed with mean $1/\mu_{local}$ time units. The rate of work due to local tasks is thus $k\lambda_{local}/\mu_{local}$. In this paper, we set $\mu_{local} = 1$, other time measures are thus relativized to the average execution time of a local task. Slack of local tasks is uniformly distributed in the range $[S_{min}, S_{max}]$.

Global Tasks

Similar to local tasks, global tasks are being generated as a *single stream* of Poisson process with mean interarrival time $1/\lambda_{global}$. In order to simplify our discussion, we hold a simple view of the world that global tasks are homogeneous. In particular, we assume that all global tasks consist of m subtasks and the execution times of the subtasks all follow the same exponential distribution with mean equal to $1/\mu_{subtask}$ time units. The total execution times of global tasks thus follow an m-stage Erlang distribution with mean $m/\mu_{subtask}$. The rate of work due to global tasks is therefore $m\lambda_{global}/\mu_{subtask}$. We assume that nodes are equally likely to be chosen as the execution node of a subtask.

We define the term *rel_flex* to be the relative flexibility of global tasks with respect to local tasks[6]. So if the distribution of slacks of global tasks is uniform in the range $[a,b]$, we can determine a and b by the following equations[7]:

$$\left(\frac{a}{m/\mu_{subtask}}\right) / \left(\frac{S_{min}}{1/\mu_{local}}\right) = rel\_flex, \qquad \left(\frac{b}{m/\mu_{subtask}}\right) / \left(\frac{S_{max}}{1/\mu_{local}}\right) = rel\_flex.$$

Or,

$$a = m \cdot rel\_flex \cdot S_{min} \cdot \frac{\mu_{local}}{\mu_{subtask}}, \qquad b = m \cdot rel\_flex \cdot S_{max} \cdot \frac{\mu_{local}}{\mu_{subtask}}.$$

A rel_flex of 1 means that global tasks and local tasks have the same flexibility distribution. A greater (smaller) than 1 rel_flex means in general, global tasks are more (less) flexible than local ones.

first.

[5]Tasks in a scheduler queue are ordered in increasing slack; The task with the least slack is served first.

[6]Recall that the flexibility of a task is defined to be the ratio of its slack over its execution time.

[7]Recall that the slack distribution of local tasks in uniform in the range $[S_{min}, S_{max}]$.

System Load

We define the *normalized load* (or *load* for short) to be the ratio of the rate of work generated to the total processing capacity of the system. That is,

$$load = \frac{\frac{m \cdot \lambda_{global}}{\mu_{subtask}} + \frac{k \cdot \lambda_{local}}{\mu_{local}}}{k}.$$

For a stable system, we have $0 \leq load < 1$.

We also define $frac\_local$ to be the fraction of *load* that is contributed by local tasks. That is,

$$frac\_local = \frac{k \cdot \frac{\lambda_{local}}{\mu_{local}}}{m \cdot \frac{\lambda_{global}}{\mu_{subtask}} + k \frac{\lambda_{local}}{\mu_{local}}}.$$

If the system does not have any local tasks, $frac\_local = 0$; a $frac\_local$ of 1 means no global tasks. As another example, if subtasks and local tasks have similar execution time, then a $frac\_local$ of 1/2 means that there are the same *number* of local tasks and global *subtasks*. However, since it takes m subtasks to make a global task, when $frac\_local$ is 1/2, there are m times more *local tasks* generated than *global tasks*.

Table 1 shows the parameter setting of our baseline experiment. In particular, we assume that the prediction on execution time is perfect, i.e., $pex(.) = ex(.)$ (In Section 7 we return to this issue and show the impact of error in the predictions.) To study the effect of these parameters on system performance, we will vary the parameters from their base settings. This is discussed in the following section.

6 Analysis

In this section, we summarize the results of our simulation experiments. As performance measure we use the percentage of missed deadlines (or miss ratio).[8] In particular, we look at the probability of a task missing its deadline conditional on its task class (i.e. global or local). We adopt the notation MD_A^B where MD stands for fraction of missed deadlines, and $A \in$ {local, global}, $B \in \{UD, ED, EQS, EQF\}$ are optional modifiers describing the task class and SDA strategy used. For example, MD_{global}^{UD} denotes the probability that a global task misses its deadline under the UD strategy.

[8] A secondary measure could be tardiness, i.e., by how much time do tasks miss their deadlines. However, due to space limitations, we do not consider this measure.

Overload Management Policy	No Abortion
Local Scheduling Algorithm	Earliest Deadline First
$\mu_{subtask}$	1.0
μ_{local}	1.0
k (# of nodes)	6
m (# of subtasks of a global task)	4
$load$	0.5
$frac\_local$	0.75
$[S_{min}, S_{max}]$	[0.25,2.5]
rel_flex	1.0
$pex(X)/ex(X)$	1.0

Table 1: Baseline setting

6.1 Baseline Experiment

As a starting point, let us look at how the various strategies do relative to each other in our baseline experiment. Figures 3(a) and 3(b) show MD_{local} and MD_{global} of the various SDA strategies as *load* varies from 0.1 to 0.5[9].

Comparing Figures 3(a) and 3(b), we see that even though global tasks and local tasks have the same average flexibility (rel_flex = 1 for the baseline setting), there is a significant difference in their ability to meet deadlines. For example, at load = 0.5, $MD^{UD}_{global} = 40\%$ (point A), while $MD^{UD}_{local} = 24\%$ (point B). However, for local tasks (Figure 3(a)) and for light loads, there is little difference.

We should point out that traditionally soft real-time systems are studied under high load situations. Most of the time, the system will hopefully operate under low load; no deadlines will be missed regardless of what scheduling policy is used. However, once in a while the system will be overloaded, and it is precisely at those times when we need a scheduling policy that can miss the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Figure 3(b) are important.

In order to understand these differences, let us consider the types of resource competition

[9] We use dotted lines for MD_{local} and solid lines for MD_{global}. We also associate symbols like $\diamond$ and $+$ to different SDA strategies.

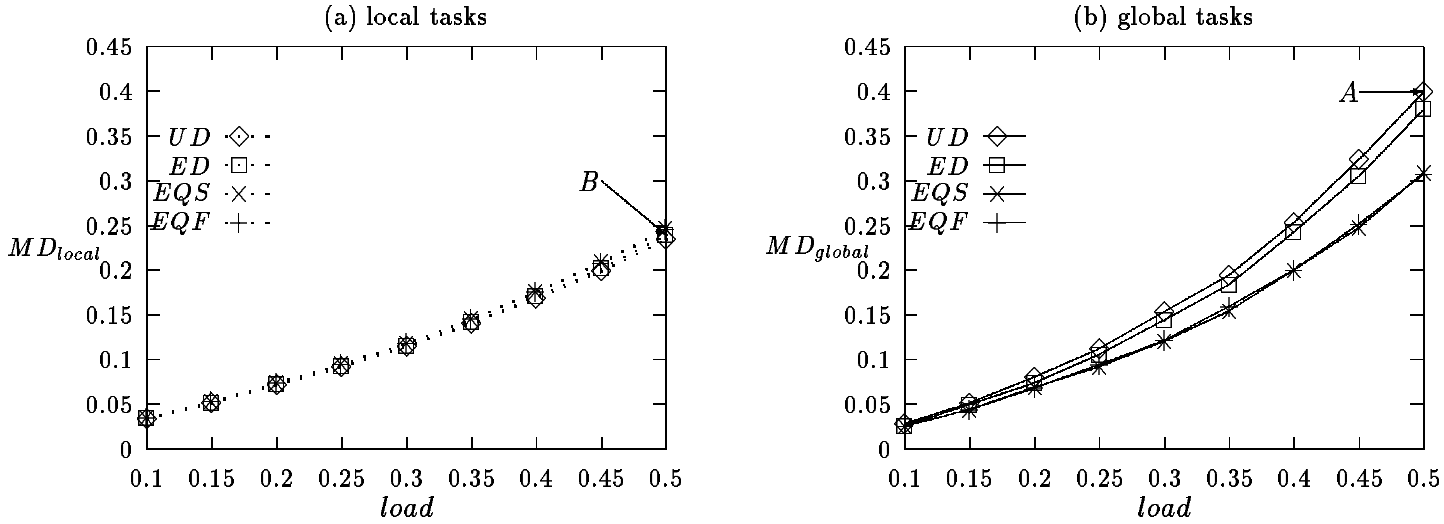

Figure 3: Performance of various SDA strategies in the baseline experiment.

among tasks. Since there are 2 task classes, there are 3 types of contention: local-local, local-global and global-global. A local scheduler resolves contention by comparing the deadlines of tasks. Since a SDA strategy only affects subtask deadlines, it only impacts local-global and global-global contention. The reason why local tasks are not affected by the SDA strategy (Figure 3(a)) is that in our baseline experiment, 75% of the load is contributed by local tasks. Thus, much of the contention faced by local tasks is local-local, and unaffected by the SDA strategy. On the other hand, global tasks face local-global and global-global contention, so the choice of a SDA strategy affects them significantly (Figure 3(b)).

Through extensive simulation experiments (not shown here), we observe (and this is confirmed in Figure 3) that the performance of ED lies between that of UD and EQF. We also observe that EQS's performance is very close to that of EQF over a wide range of parameter settings. In cases when they differ, EQF usually is superior. Thus, in order to simplify our presentation, we will exclusively focus on UD and EQF in the rest of this paper.

6.2 UD Vs EQF: Their different treatment of global tasks

To further compare UD and EQF, we overlap Figures 3(a) with 3(b), and remove the curves for ED and EQS. The result is plotted in Figure 4. From this figure, we make the following two observations:

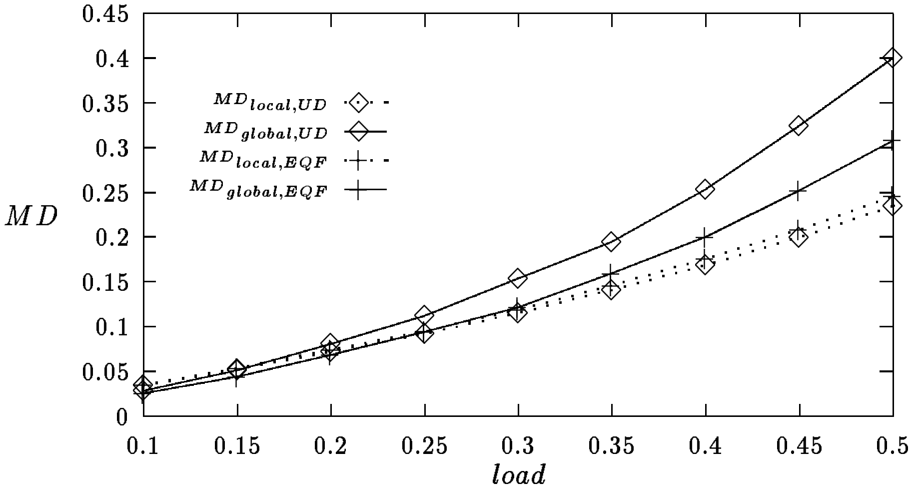

Figure 4: Comparing UD and EQF in baseline experiment.

(I) Under UD and high loads, global tasks miss many more deadlines than local tasks.

(II) Strategy EQF significantly improves the performance of global tasks (high load), but still local tasks have a better chance of meeting their deadlines.

Observation (I) is not surprising. In Section 4 we had hypothesized that UD would assign too much slack to early subtasks, resulting in a scheduling discrimination when they competed against local tasks. However, observation (II) is surprising because by assigning global tasks flexibility that was comparable to that of local tasks, we expected a "fair playing field." Unfortunately, in spite of our efforts, global tasks somehow are still hurt.

Before studying the causes of observation (II), let us confirm the hypothesis behind observation (I). Let us call this hypothesis the *discrimination hypothesis*: by giving early subtasks of global tasks too much slack, UD makes global tasks "second class citizens" as they compete with local tasks. To confirm the hypothesis, in Figure 5 we vary the relative proportion of the two task classes, i.e., we vary $frac\_local$ from 0.1 to 0.95[10].

In the figure we see that as $frac\_local$ increases (fewer global tasks), indeed MD^{UD}_{global} increases. This is because global tasks face more and more conflicts with local tasks, and are

[10]Recall that $frac\_local$ is the fraction of *load* contributed by local tasks (see definition on page 13).

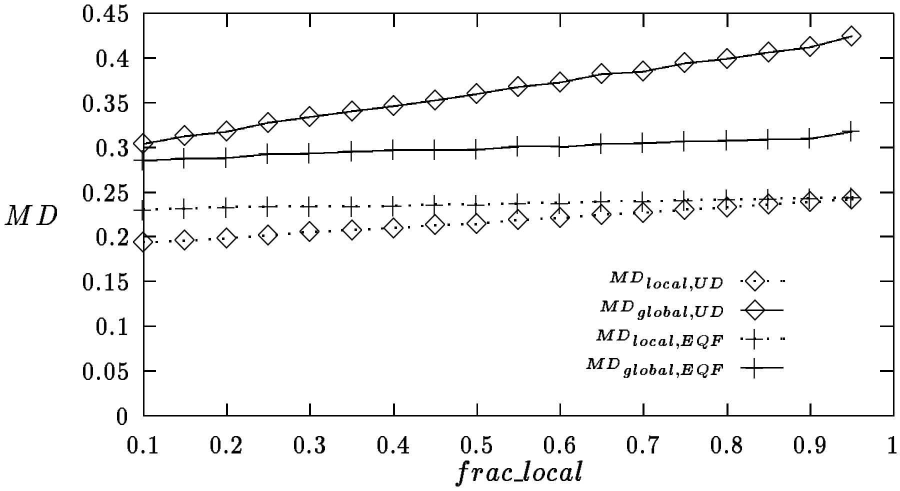

Figure 5: Effect of varying the fraction of local tasks.

discriminated against more and more. Notice that MD^{UD}_{local} also increases (although to a smaller extent): this is because local tasks are also facing more and more conflicts with "first class" tasks. On the other hand, observe that the MD^{EQF}_{local} and MD^{EQF}_{global} values hardly change as $frac\_local$ varies. This is because EQF does not discriminate against global tasks.

As a last observation on Figure 5, notice that when $frac\_local$ is close to zero (almost all tasks are global), UD and EQF perform similarly. Strategy UD is still assigning too much slack to global tasks, but since everybody is a "second class citizen," it does not really matter. Hence, in an application where most tasks are global, UD may be an adequate (and simpler) strategy.

6.3 Why $MD^{EQF}_{global} \neq MD^{EQF}_{local}$?

If there is no discrimination under EQF, then why are global tasks still performing poorly (observation (II))? The reason is that global tasks consist of a *series* of subtasks and there are two phenomena affecting the series. One phenomenon is beneficial to global tasks: if one subtask finishes early, its leftover slack is inherited by the subtasks that follow. Thus, later subtasks will tend to have even more slack. The second phenomenon is detrimental to tight tasks. If a global task has little slack to begin with, it may miss an early sub-deadline, robbing

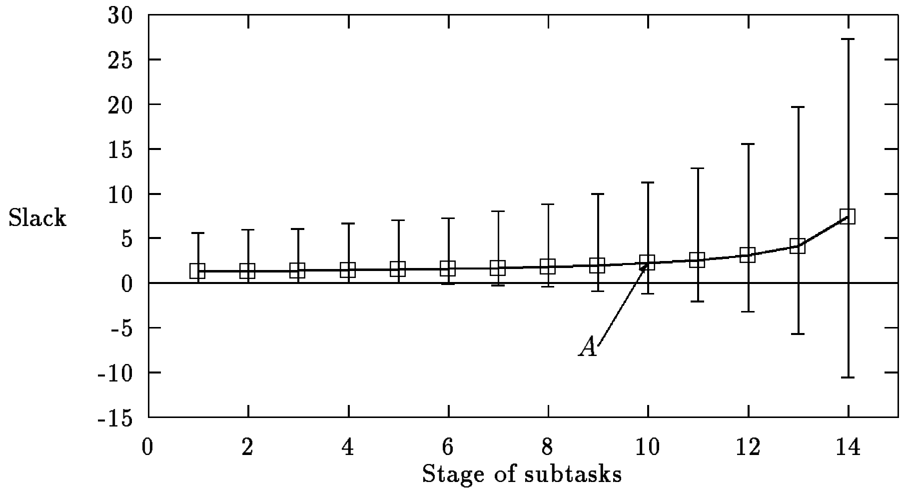

Figure 6: Average and the mid-95 percentile range of slack of subtasks versus stage of subtasks.

slack from the following subtasks, and making things even worse for them. Essentially, "the poor get poorer while the rich get richer."

To observe these phenomena, we ran an experiment with the same parameter values as our baseline experiment except that the number of subtasks of a global task (m) is now 14 instead of 4. In Figure 6, we plot the average slack of a subtask versus the stage of the subtask (the '□' line). For example, point A shows that the average slack of a 10th stage subtask is 2.3 units in our experiment. Also shown in the figure is the range of the slack data points with the largest 2.5 percentiles and the smallest 2.5 percentiles removed. The graph thus shows, for example, that the "middle" 95% of all 13th-stage subtasks have slacks in the range [-5.7, 19.7].

We can see that the average slack of a subtask increases as it advances. This is due to the beneficial phenomenon. However, the variability in slack increases, due to the detrimental phenomenon. Even though there is more slack on the average, there is a group of tasks that are suffering (the "poor") and are more likely to miss their ultimate deadlines. This is suggested by the fact that part of the 95% range bars lie below the x-axis. (As an analogy, even if the standard of living is increasing, more people may be under the poverty line.)

Depending on the system parameters, the beneficial and detrimental phenomena may have more or less force. In Figure 4, as the load increases, there are more global tasks that are "poor" to begin with (do not have much slack to spare). Thus, the detrimental effect causes

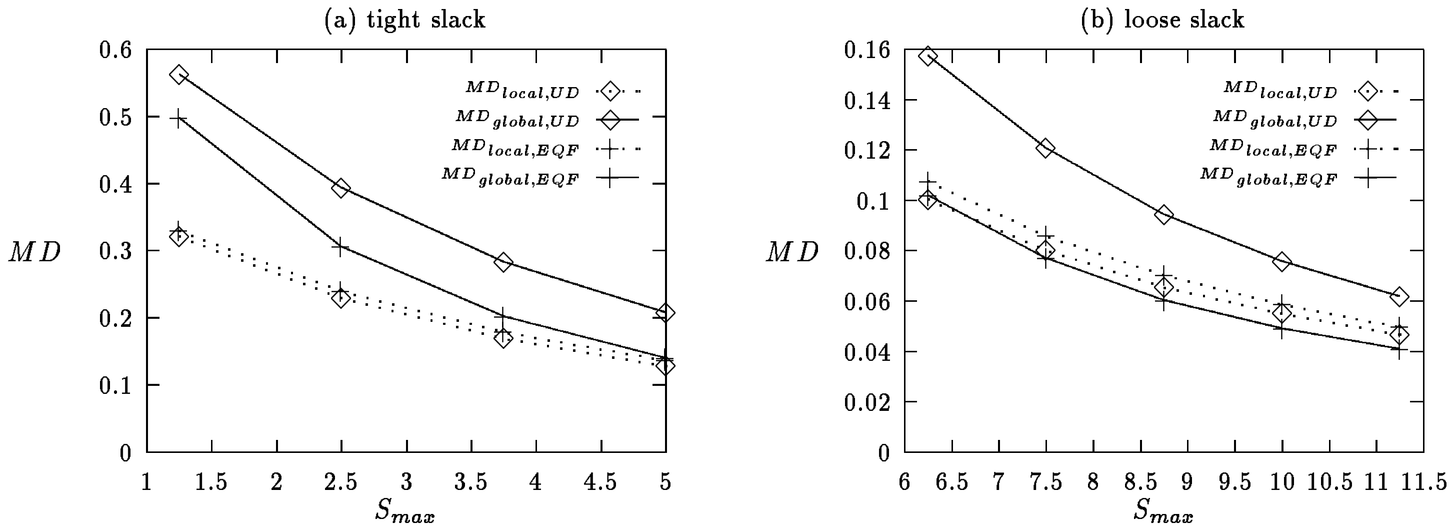

Figure 7: Effect of varying S_{max}.

more global tasks to miss their deadlines. As the load decreases, however, there are fewer poor tasks, and global tasks miss roughly the same number of deadlines as local tasks. As the load decreases further, the beneficial effect may actually cause global tasks to do *better* than local tasks. This effect can barely be seen in Figure 4, but will be seen clearly later in other graphs.

6.4 The effect of slack and the number of subtasks of a global task (m)

To evaluate the gains of EQF over UD, we varied over wide ranges all of our model parameters (Table 1). Here we report on two of the more interesting experiments, involving the slack and the number of subtasks of a global task (m).

Figure 7 shows the result of varying S_{max} from 1.25 to 11.25 for the baseline setting. To improve legibility, we divide the graph into two parts: 7(a) and 7(b), showing tight and loose slack conditions respectively. Note that these subfigures use a different MD scale.

As expected, Figure 7 shows that as slack (S_{max}) increases, tasks of all types enjoy a lower miss ratio. As in Figure 4, MD_{local} is not affected by the SDA policy (75% of the load is local tasks, and their local-local conflicts are not affected by SDA).

The choice of a SDA strategy, however, strongly affects global tasks (solid lines). When

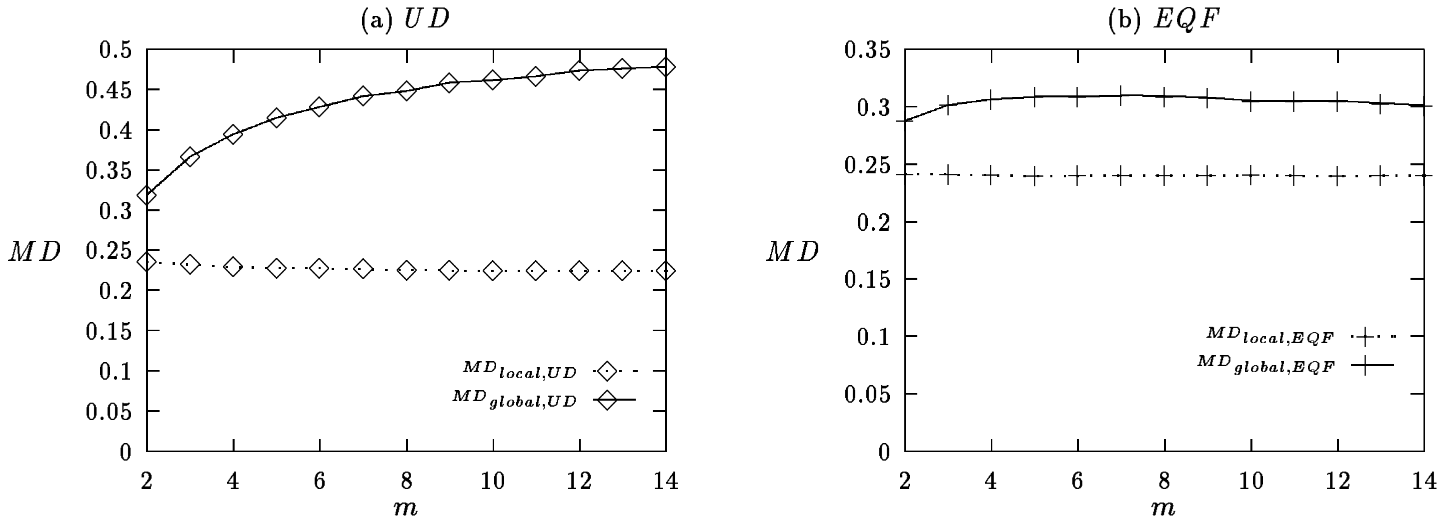

Figure 8: Effect of varying m.

slack is large (Figure 7(b)), UD consistently misses about 50% more global task deadlines than EQF does. As slack tightens (Figure 7(a)), global tasks miss more deadlines, and the gap between EQF and UD narrows, eventually closing when there is effectively no slack and all deadlines are missed.

Comparing the MD_{local}^{EQF} and MD_{global}^{EQF} curves, we observe the same effect of Section 6.3. With tight slack, there are more "poor" tasks that end up missing their deadlines because of the detrimental effect of serial subtasks. However, with more slack, the beneficial effect is stronger and we see that global tasks actually miss fewer deadlines as compared to local tasks. The crossover point occurs when $S_{max} \simeq 5$ units.

Keeping all other baseline parameters fixed, Figures 8(a) and 8(b) show the results of varying the number of subtasks, m, from 2 to 14 for UD and EQF respectively. From Figure 8(a), we see that MD_{global}^{UD} increases as m increases while MD_{local}^{UD} decreases. This shows that the discrimination problem is intensified by the length (number of subtasks) of global tasks. As a global task grows, UD incorrectly assigns more and more slack to early subtasks.

For EQF (Figure 8(b)), increasing m increases both the beneficial and the detrimental phenomena. For our parameter settings, the effects cancel each other and the result is a rather flat curve of MD_{global}^{EQF}.

To sum up, EQF always performs better than, or at least as well as, UD. The EQF gains are more significant when there is "moderate" slack and load. That is, if slack is too tight or the load too high, no matter what SDA policy we use, many deadlines will be missed. If slack is too loose or load too light, then all tasks will make their deadlines, no matter how we schedule. But in the intermediate range a smart SDA policy can make a difference and this is where EQF wins big. The EQF strategy is also superior when global tasks have many subtasks.

7 Other System Properties

In our simulation model, several assumptions are made, such as tardy tasks are not aborted, the local scheduling algorithm is earliest deadline first, and perfect execution time information is available. In this section we relax these assumptions and describe the performance implications.

Among the three assumptions listed above, the most crucial is perfect prediction of execution time. In order to study the performance sensitivity to estimate errors, we introduce "random noise" to the estimates. Specifically, for a task X we have:

$$pex(X)^{11} = \begin{cases} ex(X) * Uniform([1 - ErrorFactor, 1 + ErrorFactor]), & ErrorFactor < 1, \\ ex(X) * Uniform([0, 1 + ErrorFactor]), & ErrorFactor \geq 1, \end{cases}$$

where $Uniform([x_1, x_2])$ returns a random number uniformly distributed in the range $[x_1, x_2]$, and $ErrorFactor$ gives the *maximum* relative error in the estimation. Notice that an estimate $pex(X)$ can be either larger or smaller than the true execution time. For $ErrorFactor > 1$, the *average* estimates are biased towards overestimating the true values.

Figure 9 shows MD^{EQF}_{global} and MD^{EQF}_{local} as $ErrorFactor$ changes from 0 (no error) to 2.0. Other parameter values are kept the same as our baseline experiment. Since UD does not require the knowledge on execution time, its performance is *not* affected by $ErrorFactor$ and is omitted in the graph. (The value of MD^{UD}_{global} is a constant 0.4, significantly above MD^{EQF}_{global} for the entire experiment.)

Surprisingly, the performance of EQF is not very sensitive to errors in the execution time estimate. This is illustrated by the rather flat curves shown in Figure 9. Even if estimates are off by up to a factor of two, EQF continues to perform well. Further experimentation shows

[11] Recall that $pex()$ is the *predicted* execution time, and $ex()$ is the *real* execution time.

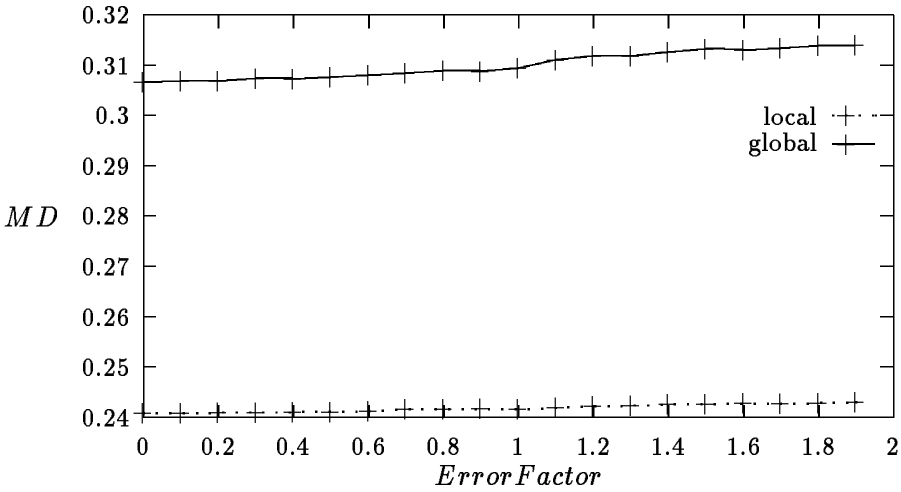

Figure 9: Sensitivity of EQF performance to $ErrorFactor$.

that this sensitivity is increased by a small amount in a high load, tight slack environment, and when *minimum-laxity-first* (see below) is used as the local scheduling algorithm.

The low sensitivity to errors may appear counter-intuitive at first, but it can be explained by two observations: (1) $ErrorFactor$ gives the maximum error, but in many cases the errors are not that far off from the true values. In all these cases, EQF makes correct deadline assignments. (2) In the cases where the estimates are significantly off, it is still beneficial to split up the available slack among subtasks, even if the split is rough. Incidentally, in a study of soft real-time database scheduling and concurrency control algorithms [2], it was also observed that errors in execution time estimates do not significantly affect performance.

Another assumption we made in the base model is that all tardy tasks are valuable and no particular action is taken when a task misses its deadline. (See Section 3.) Although not reported here in detail, we modified our simulator to abort tardy tasks. The results show that abortion helps lower the percentage of missed deadlines in general (for both local and global tasks, and regardless of the SDA policy). This is because by aborting tasks that have already missed their deadlines, we effectively reduce the workload of the system. More processing power is allocated to non-tardy tasks and this results in fewer missed deadlines.

Aborting tardy tasks does not significantly change our conclusions regarding the UD and EQF policies. However, abortion does help UD (in reducing the number of local and global

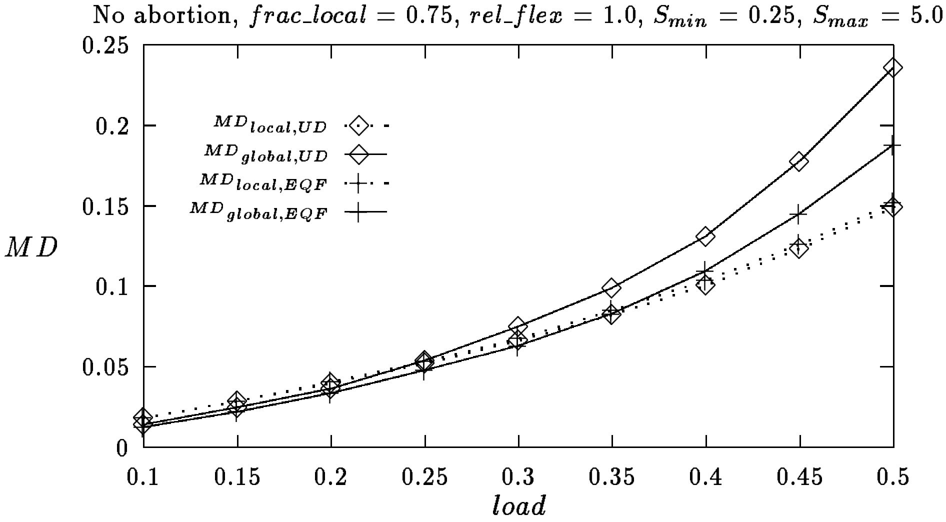

Figure 10: Performance comparison between UD and EQF with minimum-laxity-first schedulers.

task missed deadlines) more than it helps EQF. The reason is, for EQF, early-stage subtasks have less slack than later-stage subtasks (due to slack accumulation, see Figure 6). Early-stage subtasks are thus served more promptly than later-stage subtasks. A global task therefore spends relatively more time in its later stages (due to longer wait in the scheduler queues). The converse is true for UD, i.e., a global task spends more time in its early stages. Consequently, under EQF, abortion of global tasks usually happens at a later stage as compared to UD. The effective reduction of workload due to abortion is thus more significant in UD than in EQF.

Finally, we repeat the set of experiments presented in Section 6 with the earliest-deadline-first schedulers replaced by minimum-laxity-first schedulers. The results of such experiments show similar trends and relative performance between UD and EQF. As an example, Figure 10 compares the performance of UD and EQF with minimum-laxity-first schedulers and the following parameter setting: No abortion, $frac\_local = 0.75$, $rel\_flex = 1.0$, $S_{min} = 0.25$, and $S_{max} = 5.0$.

From Figure 10, we observe similar results as for the case of earliest-deadline-first schedulers. For example, at high (low) load, global tasks miss many more (fewer) deadlines than local tasks. Also, EQF significantly reduces the gap between MD_{local} and MD_{global} when the load is high.

8 Conclusions

This paper considered the problem of subtask deadline assignment in a distributed soft real-time environment. By means of extensive simulations, we compared the performance of several SDA strategies.

The main advantage of the UD (Ultimate Deadline) strategy is that only the final deadline of a global task is required for scheduling. However, our study showed that with UD global tasks miss significantly more deadlines than local tasks. An exception occurs when the system is in a low stress situation or when most tasks are global. With an estimate on subtask execution times, EQF (Equal Flexibility), on the other hand, ensures that substantially fewer global tasks miss their deadlines. This is a result of distributing the slack of a global task more "fairly" among its subasks.

We also considered other policies that give intermediate performance but require less information than EQF does. EQS (Equal Slack), for example, is useful when only the number of subtasks and the *total* remaining execution time (instead of individual subtask execution time) are known. ED (Effective Deadline), as another example, does not need to know the number of pending subtasks.

Our performance study revealed that even though EQF significantly reduces the difference between the miss ratios of local and global tasks, global tasks still miss (in many cases) more deadlines than local ones. As we pointed out, this phenomenon is due to the "multiple stages" of global tasks which induces variation in the slack distribution. An interesting modification to EQF would control the extent of slack variability, perhaps by giving subtasks of tight global tasks *less* slack than EQF would give. We intend to study this option in future research.

Finally, we would like to remark that the SDA problem is an important one in the design of *open systems*. An open system is usually built with existing standard components, often developed by different vendors. It is thus hard (or impossible) to orchestrate the independent schedulers, that are built into each individual component, to carry out a global scheduling policy. A good way of automatically assigning deadlines to subtasks which truly reflects the urgency of each unit of work is thus vital in an open system environment.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *ACM SIGMOD Record*, pages 1–12, 1988.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Procceedings of the 14th VLDB Conference*, 1988.

[3] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: a performance evaluation. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 113–124, 1990.

[4] R. Bettati and J. W. S. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Systems*, 1990.

[5] R. Bettati and J. W. S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 452–459, 1992.

[6] M. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of the 15th VLDB Conference*, pages 397–410, 1989.

[7] J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *Computing Survey*, 16(1):43–70, 1984.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[9] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[10] H. Pang, M. Livny, and M. J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of IEEE Real-Time Systems Symposiuim (to appear)*, 1992.

[11] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, 34(12):1130–1143, 1985.

[12] W. Zhao and K. Ramamritham. Virtual time CSMA protocols for hard real-time communication. *IEEE Transactions on Software Engineering*, 13(8):938–952, 1987.