The HKU Scholars Hub The University of Hong Kong 香港大學學術庫

Title	Deadline assignment in a distributed soft real-time system
Author(s)	Kao, Ben; GarciaMolina, Hector
Citation	Proceedings - International Conference On Distributed Computing Systems, 1993, p. 428-437
Issued Date	1993
URL	http://hdl.handle.net/10722/43640
Rights	Creative Commons: Attribution 3.0 Hong Kong License

Deadline Assignment in a Distributed Soft Real-Time System

Ben Kao and Hector Garcia-Molina

Abstract—In a distributed environment, tasks often have processing demands at multiple different sites. A distributed task is usually divided into several subtasks, each to be executed in order at some site. In a real-time system, an overall deadline is usually specified by an application designer indicating when a distributed task is to be finished. In this paper, we present and analyze techniques for automatically translating the overall deadline into deadlines for the individual subtasks.

Index Terms—Soft real-time, distributed systems, deadline assignment, scheduling.

1 Introduction

In traditional soft real-time applications, a task is considered a single unit of work with a given deadline. The system usually schedules tasks according to their deadlines, with more urgent ones running at higher priorities. Over the years, researchers have developed real-time scheduling algorithms for different real-time system components, including the communication network [8], [12], [5], database [1], disk I/O [2], and processor [9]. One common tacit assumption made by these algorithms is that the deadline of a task truly reflects the urgency of completing the task. As real-time systems evolve, however, "tasks" become "bigger," more complicated, and more frequently possess subtasks to be executed on various system nodes or components. In a distributed environment, local schedulers find themselves scheduling subtasks, or "segments" of global tasks, instead of complete, integrated tasks. In most situations, a single value of an end-to-end global deadline fails to capture the sense of urgency of each individual subtask. This severely hampers the efficacy of real-time scheduling algorithms.

As an example of a complex distributed task, let us consider stock market analysis and program trading. In this application, information on stock prices is gathered through multiple sources and is piped through a series of filters for refinement. The information is then used by an expert system that spots trading opportunities. This latter stage may involve extensive database operations and knowledge rule processing. A profit may then be realized by the appropriate buy and sell actions. While the deadlines for high-level tasks are usually given as a part of the system specification (e.g., a buy-sell action should be implemented within two minutes from the time when the information is gathered), we lack a methodical way of assigning deadlines to the individual subtasks (e.g., how much time should we give a database search? a disk access? a network transmission?). In this paper, we study the subtask deadline assignment problem (SDA), and suggest guidelines for deriving subtask deadlines from a global task's end-to-end deadline. Our study is abstract in nature, trying to identify the broad classes of strategies that can be used and their general implications. Our goal is not to present concrete algorithms or performance results for a particular system.

To study the SDA problem, we need to understand the structure of global tasks. A global task can be very complex, with arbitrary precedence relationships among its subtasks. Many global tasks, however, fall into the category of serial-parallel tasks, which have a simpler structure. For this type of tasks, we can generally reduce the SDA problem into two simpler subproblems: the serial subtask problem (SSP, Section 4) and the parallel subtask problem (PSP, Section 5). In each case, we assign deadlines to the serial or parallel subtasks that make up the task. We can then combine these results for tasks that have both serial and parallel components (Section 6).

In this paper, we focus on *soft real-time* systems (although we would like to remark that the techniques we will discuss can also be applied to hard systems). We assume that the distributed system consists of independent *components*, each with its own scheduler. The schedulers do not perform load balancing among them. 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.

2 Related Work

There are relatively few studies on the SDA problem [6]. However, we would like to mention two studies that are closely related to our approach and SDA. Bettati and Liu [3], [4] discuss 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

- *B. Kao is with the Department of Computer Science, The University of Hong Kong. E-mail: kao@cs.hku.hk.*
- *H. Garcia-Molina is with the Department of Computer Science, Stanford University, Stanford, CA 94305-9040. E-mail: hector@cs.stanford.edu.*

Manuscript received 29 Apr. 1993.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100593.

1045-9219/97/$10.00 © 1997 IEEE

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 [11], by Pang et al., 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.

Our work is similar to [11] 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 The Model

In this section, we describe the task and system model we use to study the SDA problem. We will first define global tasks, and, then, describe a simple model of a distributed system on which tasks are mapped for execution. We will also define some terms that will help in our discussion.

3.1 The Task Model

We consider two types of tasks in our system: locals and globals. A local task is one that is executed at one and only one node. (Each system component, e.g., a database server, is represented by a node in our model.) A global task, on the other hand, can be quite complex and may involve work at multiple nodes in the system. In this paper, we only consider global tasks that are serial-parallel. As shorthand, we use the notation $T = [T_1\ T_2\ ...\ T_n]$ to represent a *global* task T that consists of n subtasks, $T_1, T_2, ..., T_n$, to be executed in *series*. A subtask T_i ($i > 1$) cannot execute before subtask T_{i-1} finishes. We also use the notation $T = [T_1 \parallel T_2 \parallel ... \parallel T_n]$ to represent a global task T consisting of n subtasks $T_1, T_2, ..., T_n$ to be executed in *parallel*. The n subtasks arrive at the same time and task T is considered finished only if all n subtasks finish. Composition of these notations is possible. We call a subtask which involves execution only at a single node a *simple subtask*. A subtask that is itself a global task is called a *complex* subtask.

A task X (whether it is a local task, a simple subtask, or a global task) has the following five attributes: arrival time ($ar(X)$), deadline ($dl(X)$), slack ($sl(X)$), *real* execution time ($ex(X)$), and *predicted* execution time ($pex(X)$).

We do not assume the value of $ex(X)$ be available, but some of our SDA strategies do take advantage of an estimate, $pex(X)$, which is an approximation to $ex(X)$. These attributes are related by: $dl(X) = ar(X) + ex(X) + sl(X)$.

We also define flexibility (denoted by $fl(.)$) of a task X to be the ratio 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.

Finally, in this paper, tardy tasks are not aborted. (For the firm deadline policy, see [6], [7].)

3.2 The System Model

Our model of a distributed real-time system consists of a number of *nodes* representing different processing components (Fig. 1). These nodes manage different resources, like a database, an expert system, or a compute engine. Even the communication network is considered a resource and is subsumed as one or more processing nodes. Each node services both local tasks (which are generated at each node), as well as simple subtasks of global tasks. Task service order is scheduled by a real-time scheduler residing at each node. These schedulers are all independent and they do not collaborate. The only things that influence scheduling decision are the real-time attributes associated with each task.

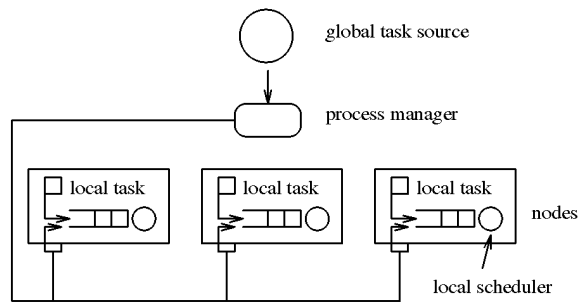


Fig. 1. The system model.

Newly created global tasks are first processed by the process manager. We assume that certain control information of a global task, such as the precedence relationship among the subtasks and the end-to-end deadline, is available to the process manager. The major functions of the process manager are to assign deadlines to simple subtasks, submit the simple subtasks to the appropriate nodes for execution, and enforce the precedence constraints among the subtasks of a global task. In reality, the process manager consumes system resources (e.g., communication overhead between the manager and the nodes,) but, as we pointed out earlier, this consumption can be considered as additional subtasks that can be handled similarly by the SDA algorithm. We therefore do not explicitly model the resource requirement of the process manager.

4 The Serial Subtask Problem (SSP)

As we mentioned earlier, the SDA problem can be divided into two subproblems: SSP and PSP for the class of serial-parallel tasks. Our approach to SDA is to study the two subproblems individually, and, then, combine the techniques to devise an integrated SDA strategy. In this section, we study the serial subtask problem (SSP). We suggest

some possible solutions and evaluate them through simulation. The results of the analysis will be presented in Section 4.2.

To study SSP, we consider global tasks of the form: $T = [T_1\ T_2 \ldots T_m]$ where the T_is ($1 < i < m$) are all *simple subtasks*. That is, a global task only consists of a sequence of subtasks to be executed *in order*. An SSP strategy is one that determines the values of a *virtual deadline* $dl(T_i)$ ($1 \le i \le m$) at the time when T_i is submitted.

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 SSP strategy would be to set the deadline of a subtask to be equal to the deadline of its global task. We call this strategy *Ultimate Deadline* (*UD*).

(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 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 minus the total expected execution time of the subtasks of T following T_i. 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 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.

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 amounts 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 equally among the *remaining* subtasks. This gives us the *Equal Slack (EQS)* strategy:

(3) Equal Slack (EQS):

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

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):

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

4.1 Simulation Model

In order to study and contrast system behavior under these SSP strategies, we developed a simulation model and performed extensive experiments. In this section, we describe the model; our results are presented in Section 4.2.

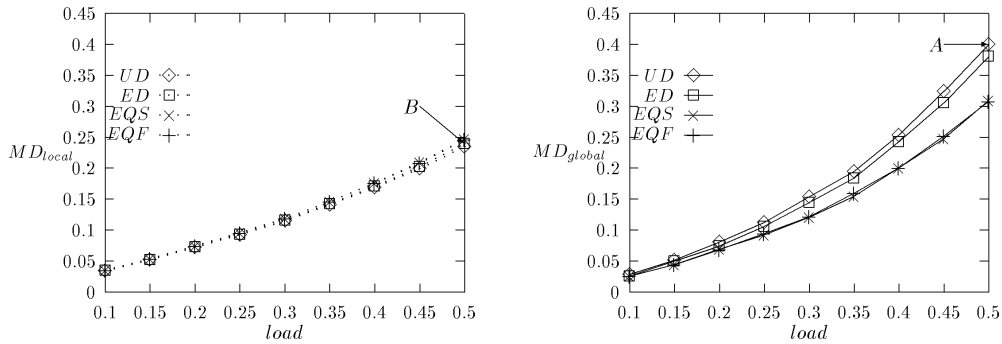
Our simulator is written in the simulation language **DeNet** [10]. Each simulation experiment (generating one data point) consists of two simulation runs, each lasting one million time units (at least 100,000 tasks are generated per run, many more for high load experiments). The 95 percent confidence interval is ±0.35 percentage points (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* as the scheduling algorithm for most of our experiments (see [6] for other scheduling algorithms).

Local Tasks. Local tasks are being generated *at each node* according to a Poisson distribution with mean interarrival time $1/\lambda_{local}$ time units. (Poisson distributions are typically used in analytical studies like ours because of their simplicity and because they yield useful insights.) 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 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



(a) (b)

Fig. 2. Performance of various SSP strategies in the baseline experiment: (a) local tasks, (b) global tasks.

to global tasks is therefore $m\ \lambda_{global}/\mu_{subtask}$. The execution node of a subtask is picked randomly (and uniformly) from the k nodes. We also define the term *rel_flex* to be the relative flexibility of global tasks with respect to local tasks.

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,

$$load = \left(\frac{m \cdot \lambda_{global}}{\mu_{subtask}} + \frac{k \cdot \lambda_{local}}{\mu_{local}} \right) \Big/ k .$$

For a stable system, we have $0 \le load < 1$. We also define *frac_local*} to be the fraction of *load* that is contributed by local tasks. That is,

$$frac_local = \left(k \cdot \frac{\lambda_{local}}{\mu_{local}} \Big/ \left(m \cdot \frac{\lambda_{global}}{\mu_{subtask}} + k \frac{\lambda_{local}}{\mu_{local}} \right) \right).$$

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*(.). (Readers are referred to [6] for a discussion on "error in the execution time 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.

4.2 Results

In this section, we summarize the results of our simulation experiments on SSP. The primary performance measure we use to evaluate our algorithms is the percentage of missed deadline (or miss ratio). 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 SSP strategy used. For example, MD_{global}^{UD} denotes the probability that a global task misses its deadline under the *UD* strategy.

TABLE 1
BASELINE SETTING

Overload Management Policy	No Abort
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

4.2.1 Baseline Experiment

As a starting point, let us look at how the various strategies do relative to each other in our baseline experiment. Figs. 2a and 2b show MD_{local} and MD_{global} of the various SSP strategies as *load* varies from 0.1 to 0.5.

Comparing Figs. 2a and 2b, 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_{global}^{UD} = 40% (point *A*), while MD_{local}^{UD} = 24% (point *B*). Also, different SSP strategies miss different numbers of global task deadlines, unless the load is very light (Fig. 2b).

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 that we need a scheduling policy that can miss the fewest deadlines. For this reason, the big differences in missed deadlines under high load in Fig. 2b are important.

In order to understand these differences, let us consider the types of resource competition among tasks. Since there are two task classes, there are three types of contention: local-local, local-global, and global-global. A local scheduler resolves contention by comparing the deadlines of tasks. Since an SSP strategy only affects subtask deadlines, it only impacts local-global and global-global contention. The reason why local tasks are not affected by the SSP strategy (Fig. 2a) is that, in our baseline experiment, 75 percent of the load is contributed by local tasks. Thus, much of the contention faced by local tasks is local-local, and unaffected

by the SSP strategy. On the other hand, global tasks face local-global and global-global contention, so the choice of a SSP strategy affects them significantly (Fig. 2b).

Through extensive simulation experiments (not shown here), we observe (and this is confirmed in Fig. 2) 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 section.

4.2.2 UD Vs EQF: Their Different Treatment of Global Tasks

Comparing the miss rates of *UD* and *EQF*, we make the following observations:

1) Under *UD* and high loads, global tasks miss many more deadlines than local tasks.
2) Strategy *EQF* significantly improves the performance of global tasks (high load), but, still, local tasks have a better chance of meeting their deadlines.

Observation 1 is not surprising. We had hypothesized that, 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 Fig. 3, we vary the relative proportion of the two task classes, i.e., we vary *frac_local* from 0.1 to 0.95.

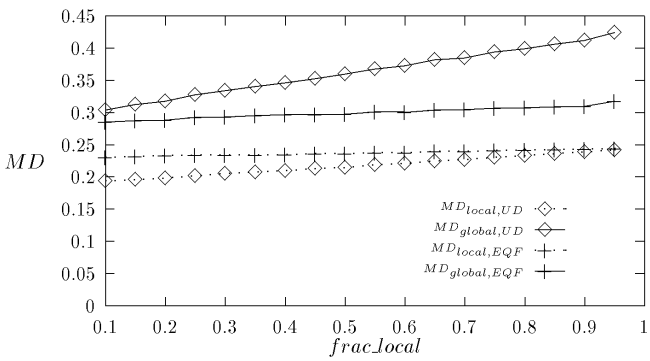


Fig. 3. Effect of varying the fraction of local tasks.

In the figure, we see that, as *frac_local* increases (fewer global tasks), indeed MD_{global}^{UD} increases. This is because global tasks face more and more conflicts with local tasks, and are discriminated against more and more. Notice that MD_{local}^{UD} 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_{local}^{EQF} and MD_{global}^{EQF} values hardly change as *frac_local* varies. This is because *EQF* does not discriminate against global tasks.

If there is no discrimination under *EQF*, then why are global tasks still performing poorly (Observation 2)? 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 subdeadline, robbing slack from the following subtasks, and making things even worse for them. Essentially, "the poor get poorer while the rich get richer."

4.3 Variations of the Baseline Model

To evaluate the gains of *EQF* over *UD*, we varied all of our model parameters over wide ranges, for example, the slack and the number of subtasks of a global task. Our observation is that *EQF* almost 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 SSP 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 SSP policy can make a difference and this is where *EQF* wins big. The *EQF* strategy is also superior when global tasks have many subtasks [6].

Several assumptions have been made in the baseline experiments so far. In particular, we have assumed that tardy tasks are not aborted, the local scheduling algorithm is earliest-deadline-first, execution time estimates are perfect, and information on the number of subtasks is available. We have conducted extensive experiments in which these assumptions are relaxed. In particular, we have studied the cases in which random error is introduced into the task execution time estimate, in which tardy tasks are aborted, and in which minimum-laxity-first is used as the local scheduling algorithms. In addition, we studied a scenario when global tasks can have different number of subtasks, and another scenario where some of the nodes had higher local task loads than others. Due to space limitations, we do not include results for all these cases here. However, the results do not change the basic conclusions presented in the previous sections.

5 The Parallel Subtask Problem (PSP)

In this section, we discuss the second part of the SDA problem: PSP. To study PSP, we only consider global tasks of the form: $T = [T_1 \parallel T_2 \parallel ... \parallel T_m]$, where the T_is $(1 < i < m)$ are all *simple subtask*. For the global task T to meet its deadline ($dl(T)$), all T_is have to be finished before $dl(T)$, their *natural* deadline.

In a soft real-time environment, when a task is submitted to a node for execution, there is no guarantee that the task will be completed before its deadline. There is, thus, a probability that a task becomes tardy due to a transient overload at its execution node. This "missed deadline" probability gets amplified in the case of global tasks with parallel subtasks because, if *any* subtask misses the deadline, the whole group becomes tardy.

5.1 Heuristics for PSP

To give global tasks a better chance of completing, we can again assign their subtasks *virtual* deadlines before they are submitted to their execution nodes. With our global task $T =$

$[T_1 \parallel T_2 \parallel ... \parallel T_n]$, our goal is to set a virtual $dl(T_i)$ from $dl(T)$.

As a base strategy for comparison, we set $dl(T_i) = dl(T)$. That is, the subtasks inherit the deadline of their global task. We call this the *Ultimate Deadline* strategy (*UD*): $dl(T_i) = dl(T)$.

To make the simple subtasks of global tasks more competitive, we need to set their deadlines earlier. Here, we look at a class of strategies called *DIV-x*:

$$\textbf{DIV–x: } dl(T_i) = [dl(T) - ar(T)] / (n * x) + ar(T). \quad (1)$$

Here, x is a parameter we can adjust. The *DIV-x* strategy simply divides the amount of time that a global task has by x times its number of subtasks. The larger the value of x is, the earlier are the virtual deadlines assigned to the subtasks, and, thus, the higher the priority of the subtasks.

One may notice that, with the *DIV-x* strategy, the virtual deadlines assigned to the subtasks are, however big x is, later than the tasks' arrival time. A subtask, therefore, may still have a lower priority than a local task if the local task has an early enough deadline. A strategy that is even more aggressive than *DIV-x* would always serve subtasks before locals. We call this strategy *Globals First* (*GF*). With *GF*, the earliest-deadline-first servicing order is preserved individually within the classes of globals and locals. However, global subtasks are always scheduled before local tasks.

5.2 Simulation Model

We use the same simulation model as in the SSP case, except that global tasks now consist of purely parallel subtasks. Specifically, a global task T consists of m subtasks $T_1, T_2, ..., T_m$ to be executed in parallel at m *different* nodes (we use the same value m for all global tasks). The deadline of a global task is set by the following formula:

$$dl(T) = \max_i \{ex(T_i)\} + slack + ar(T). \quad (2)$$

where $\max_i\{ex(T_i)\}$ is the execution time of the longest subtask among the T_is, and *slack* is the slack chosen (from the uniform distribution) for this particular global task. We note that, even though the slack of global tasks and local tasks is generated from the same slack distribution, on average, a *subtask* of a global task has more slack than a local. Also, we use the same baseline setting (see Table 1), except that the slack distribution is now [1.25, 5.0].

5.3 Results

UD. Fig. 4 compares the performance of *UD* (□) and *DIV-x* for $x = 1$ (◇) and $x = 2$ (×). Let us first focus on *UD* and *DIV-1*. The x-axis is the normalized load to the system, while the y-axis shows the fraction of missed deadlines of the various task types. As the load increases, the waiting time of tasks increases and more tasks (of all kinds) miss their deadlines.

From the figure, we see that *UD* causes global tasks to miss their deadlines almost three times as often as locals. In general, it is inadequate to assign the deadline of a global task to its subtasks and let them compete fairly with local tasks.

DIV-x. From our previous discussion, we can deduce that the more subtasks a global task has, the poorer is its chance of meeting its deadline. By dividing up the amount of time that a global task is allowed to finish (see (1)), *DIV-x* effectively promotes the priority of the subtasks and, thus, reduces global task miss rate. One nice property of *DIV-x* is that the amount of priority promotion grows with the number of subtasks of the global task. It, therefore, adjusts automatically to the need.

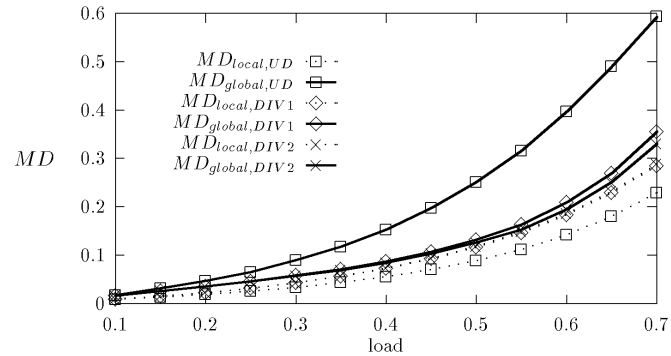


Fig. 4. Performance of *UD* and *DIV-x* in baseline experiment.

By giving subtasks higher average priority, *DIV-1* manages to keep the miss rate of both locals and globals at similar level (the two ◇ lines are close to each other). Since only the subtasks are given earlier virtual deadlines for a raise in their priority, local tasks suffer from this unfairness with a higher miss rate than under *UD*. However, under our baseline setting, this increment is marginal compared with the improvement achieved on global tasks.

By pushing the virtual deadlines of subtasks further earlier, *DIV-2* raises the priority of subtasks even higher than does *DIV-1*. The difference between their performance, however, is hardly noticeable, except at very high load. Setting $x > 1$ in our baseline experiment is, therefore, not necessary to provide a low level of missed deadlines for global tasks. The question of how to set the value of x for the *DIV-x* strategy is addressed in [7].

GF. The minute difference between *DIV-1* and *DIV-2*, as shown in Fig. 4, may suggest that one should not look further for even more aggressive strategies. *GF*, which represents the ultimate one in raising subtask priority, may not be expected to provide any significant improvement over *DIV-x* in reducing global task miss rate. Surprisingly, our experiment shows that *GF does* further reduce MD_{global} by a significant amount. One disadvantage of *GF*, however, is that it is not applicable to components that discard tasks with a *past* deadline (virtual or not). Due to space limitation, we refer readers to [7] for a discussion of the *GF* policy.

As a conclusion, our baseline experiment shows that the PSP problem can be corrected at the expense of losing some local tasks. Two simple strategies *DIV-x* and *GF* are shown to be effective under our baseline setting. For additional results on the PSP problem, readers are referred to [7].

6 SSP + PSP

The SSP and PSP strategies discussed in this paper can be integrated nicely and be applied to serial-parallel tasks: A global deadline is broken down into virtual deadlines using either the SSP or the PSP strategies, depending on whether the global task is serial or parallel. If a subtask is itself a

complex serial-parallel task, the virtual deadline assigned to it is further decomposed. For example, if, at the highest level, the task consists of serial subtasks, we first use SSP. If, say, the first of these subtasks consists of parallel subtasks, then we apply PSP to it, and so on.

To study the relative importance of the SSP and PSP strategies and how they affect complex distributed tasks, we ran experiments on a system of serial-parallel tasks with four SDA strategies applied, namely, *UD-UD*, *UD-DIV1*, *EQF-UD*, and *EQF-DIV1* corresponding to the different SSP-PSP combinations. Our observation is that the *UD-UD* strategy misses vastly more global deadlines than it misses local ones. The application of either *EQF* or *DIV-1* significantly reduces MD_{global} with a mild increment in MD_{local}. Also, the two strategies complement each other and, when applied at the same time, are able to keep MD_{global} close to MD_{local}, even under a high load situation. This suggests that the SSP and the PSP policies can be combined, and that their benefits are "additive." As soft real-time applications get larger and more complex, our results show that a good SDA strategy becomes a very crucial part of the system design.

7 Conclusion

For the class of serial-parallel tasks, the SDA problem can be divided into two subproblems: SSP and PSP. For SSP, 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. One trick would be to add artificial stages. We intend to study this option in future research.

For PSP, *DIV-x* and *GF* are two effective strategies. Our study shows that they are most outstanding under high load situation and when there is a nontrivial population of local tasks in the system. Between *DIV-x* and *GF*, *GF* usually holds an edge if tardy task abort is not supported by the system. Otherwise, *DIV-x* is a better choice because it evens up the miss rate of global tasks with different number of subtasks.

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," *SCM SIGMOD Record*, pp. 1-12, 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proc. IEEE Real-Time Systems Symp.*, pp. 113-124, 1990.

[3] R. Bettati and J.W.S. Liu, "Algorithms for End-to-End Scheduling to Meet Deadlines," *Proc. Second IEEE Conf. Parallel and Distributed Systems*, 1990.

[4] R. Bettati and J.W.S. Liu, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proc. IEEE Real-Time Systems Symp.*, pp. 452-459, 1992.

[5] D.D. Kandlur, K.G. Shin, and D. Ferrari, "Real-Time Communication in Multi-Hop Networks," *Proc. 11th Int'l Conf. Distributed Computing Systems*, pp. 300-307, 1991.

[6] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," Technical Report STAN-CS-92-1452, Stanford Univ., 1992.

[7] B. Kao and H. Garcia-Molina, "Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks," *Proc. 14th Int'l Conf. Distributed Computing Systems*, 1994.

[8] J.F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-Access Protocols and Time-Constrained Communication," *Computing Survey*, vol. 16, no. 1, pp. 43-70, 1984.

[9] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, 1973.

[10] M. Livny, "**DeNet** User's Guide," technical report, Univ. of Wisconsin-Madison, 1990.

[11] H. Pang, M. Livny, and M.J. Carey, "Transaction Scheduling in Multiclass Real-Time Database Systems," *Proc. IEEE Real-Time Systems Symp.*, 1992.

[12] W. Zhao and K. Ramamritham, "Virtual Time CSMA Protocols for Hard Real-Time Communication," *IEEE Trans. Software Eng.*, vol. 13, no. 8, pp. 938-952, Aug. 1987.

Ben Kao received the BS degree in computer science from the University of Hong Kong in 1989 and the MS degree in computer science from Princeton University, Princeton, New Jersey, in 1991. He is currently an assistant professor in the Department of Computer Science at the University of Hong Kong. From 1989-1991, he was a teaching and research assistant at Princeton University. From 1992-1995, he was a research fellow in the Computer Science Department at Stanford University, Stanford, California. His research interests include database management, distributed algorithms, real-time systems, and information retrieval systems.

Hector Garcia-Molina received a BS in electrical engineering from the Instituto Tecnologico de Monterrey, Mexico, in 1974, and an MS in electrical engineering and a PhD in computer science from Stanford University, Stanford, California, in 1975 and 1979, respectively. Dr. Garcia-Molina is currently the Leonard Bosack and Sandra Lerner Professor in the Department of Computer Science and Electrical Engineering at Stanford University. From 1979 to 1991, he was a member of the faculty of the Computer Science Department at Princeton University, Princeton, New Jersey. His research interests include distributed computing systems and database systems. Dr. Garcia-Molina is a fellow of the ACM.