Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks

Ben Kao* Hector Garcia-Molina†

Abstract

Complex distributed tasks often involve *parallel* execution of subtasks at different nodes. To meet the deadline of a global task, all of its parallel subtasks have to be finished on time. Comparing to a local task (which involves execution at only one node), a global task may have a much harder time making its deadline because it is fairly likely that at least one of its subtasks run into an overloaded node. Another problem with complex distributed tasks occurs when a global task consists of a number of *serially* executing subtasks. In this case, we have the problem of dividing up the end-to-end deadline of the global task and assigning them to the intermediate subtasks. In this paper, we study both of these problems. Different algorithms for assigning deadlines to subtasks are presented and evaluated.

Keywords: soft real-time, distributed systems, parallel 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 time by which the task should be completed. 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 [7, 14], database [1], disk I/O [2], and processor [8]. 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 the well-designed real-time scheduling algorithms.

*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

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 processing knowledge rules. 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 2 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 on how subtask deadlines are derived from a global task's end-to-end deadline.

To study the SDA problem, first of all, 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 (we will define serial-parallel tasks later). For this type of tasks, we can generally reduce the SDA problem into two simpler sub-problems: one deals with serial subtasks, and another one with parallel subtasks. To illustrate the concept, let us consider the following example:

$$T : \left\{ \begin{array}{c} T_{11} \\ .. \\ T_{15} \end{array} \right\} T_2 \qquad \text{deadline of } T = 10.$$

Here, we have a global task T which consists of two serial stages: $\{T_{11}, .., T_{15}\}$, and T_2. The first stage is a parallel task with five subtasks. Each of the six subtasks is to be scheduled at a different system component (e.g., disks, networks, processors). Assuming task T arrives at time 0 and has a deadline at time 10, what deadlines should we assign to the six subtasks? If we use the end-to-end deadline (time 10) as the subtasks' deadlines, a scheduler will be fooled to believe that it has 10 time units to complete the first stage $\{T_{11}, ..., T_{15}\}$. This may leave very little or no slack for subtask T_2, and a missed deadline may ensue. An earlier deadline for the first stage should therefore be used instead, but how early should it be? What happens if we have three serial stages instead of two? We refer the problem of breaking up an end-to-end deadline into earlier ones for serial subtasks as the *serial subtask problem* (SSP).

Now suppose we figure out that we should reserve at least 5 seconds for T_2 in order to have a good chance of making the deadline. In this case, we would like the first stage be done by time 5. It seems natural to assign 5 as the deadline for each of T_{11}, ..., T_{15}. However, this turns out to be inadequate. As an analogy, consider a group of friends agreed on meeting at a theatre for a movie. If that is a small group of one or two people, they will probably all arrive before the show. However, if the group consists of 10 members, chances are that someone will show up late and the whole group misses the movie. A similar problem occurs in our example: when a real-time task is being divided up into a number of subtasks for parallel processing, it is very likely that at least one of the subtasks run into a busy component and become tardy. This will cause the whole global task (in our example, the first stage) to miss its deadline (time 5). The *parallel subtask problem* (PSP) deals with assigning deadline to parallel subtasks.

The serial subtask problem has been studied previously in [6]. We will present a summary of the results later in Section 8. The goal of this paper is to study the parallel subtask problem and the combined effect of PSP and SSP. Specifically, we suggest and evaluate two heuristic scheduling policies for dealing with PSP. Working together with the subtask deadline assignment strategies proposed in [6], we show that the real-time behavior of complex distributed tasks can be significantly improved. Before we proceed, we state some premises our study is based on:

- We focus on *soft real-time* systems. In such systems, a primary performance goal is to meet as many deadlines as possible, but unlike hard real-time systems, there is no absolute guarantee that all deadlines will be met. There are two reasons why we look at soft (instead of hard) real-time systems. Firstly, the kind of distributed tasks we are looking at are quite complex, involving multiple stages and parallel processing. This means it is generally hard to get a reasonably accurate running time estimate for hard real-time scheduling. Secondly, in many applications it is undesirable or impossible to place an upper bound on the load. This makes hard real-time scheduling impossible to achieve.

- The scheduler at each component is independent of the others. There is no global scheduler that instructs each 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.

- We look at *on-line* scheduling, as opposed to a-priori when the tasks are defined or first submitted. On-line assignment is superior in soft real-time systems, where the types of tasks or their durations may be unknown in advance. Also, when the system provides distribution transparency (e.g., whether a piece of data item is available locally or remotely), the subtasks to be created is unknown until run-time and thus off-line pre-analysis is not appropriate.

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

We start in Section 2 by surveying some related work. Section 3 describes our distributed system model and task model. In Section 4, we take a closer look of the parallel subtask problem, and suggest some possible solutions. Section 5 contains the details of our simulation model, while Sectins 6 and 7 present the results of our simulation study on PSP. Section 8 summarizes the results found in [6] on SSP. We also combine the PSP and SSP strategies and study their performance benefit by applying them to a typical distributed application. Finally, we conclude our paper in Section 9.

2 Related Work

The problem of scheduling real-time distributed tasks has been studied mostly in a hard real-time environment (e.g., see [5, 11, 4, 12, 13, 3]). Some of these works concentrate on the task-allocation/load-balancing strategies, and on the schedulability of tasks. The communication overhead between subtasks with precedence relationship is considered when a task allocation decision is made. For example, subtasks that communicate a lot are usually scheduled to be executed at the same node. Information about tasks, e.g., their (worst case) execution time, is assumed known in advance. In these studies, system nodes usually have similar capability and they share the load. Scheduling is done in an orchestrated fashion. On the other hand, our study focuses on *open systems* which are built out of pre-existing components of different nature. A database server, for example, will not spare its cycles to help out an image processing node. Global scheduling and load balancing are therefore not appropriate in this environment.

There are relatively few studies on the SDA problem [6]. However, in [10], a problem related to SSP is addressed. In their study, Pang et. al. investigate 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 because of their "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 designed to eliminate the bias phenomenon. The SSP problem can be seen as a discrete version of the one studied in [10]. A global task with a number of serial subtasks can be considered as a long transaction. Their method of assigning virtual deadlines can also be used to assign the subtask deadlines.

3 The Model

In this section, we describe the task model and the system model we use to study PSP and SSP. 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 us 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. A global task, on the other hand, can be quite complex and may involve work at multiple components 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 $T = [T_1 || T_2 || ... || T_n]$ 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. For example, $[T_1 [T_{21} || T_{22}] T_3]$ represents a global task that has three subtasks to be run in series, and the second subtask is itself a global task of two parallel subtasks (T_{21} and T_{22}).

Formally, we define the class of (serial-parallel) global task by the following recursive rules:

GT1: A single task which is part of a larger task and which is to be executed at one and only one node is called a *simple subtask*.

GT2: If $T = [T_1 T_2 ... T_n]$ where T_i's are either simple subtasks or global tasks, then T is a global task.

GT3: If $T = [T_1 || T_2 || ... || T_n]$ where T_i's are either simple subtasks or global tasks, then T is a global task.

For rules GT2 and GT3, if T_i is itself a global task, we say that T_i is a *complex subtask* of task T. Figure 1 illustrates the various terms and concepts.

A task X (whether it is a local task, a simple subtask, or a global task) is associated with the following attributes:

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

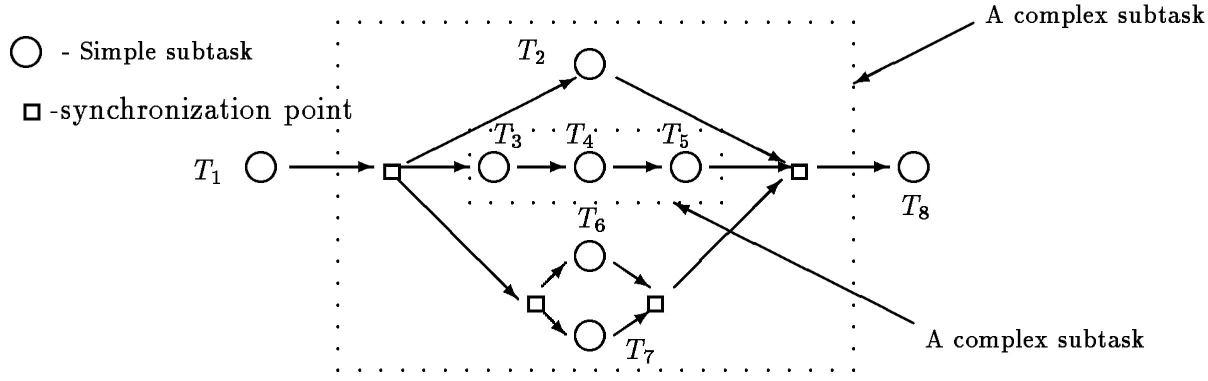

Figure 1: A global task example: $[T_1[T_2||[T_3T_4T_5]||[T_6||T_7]]T_8]$.

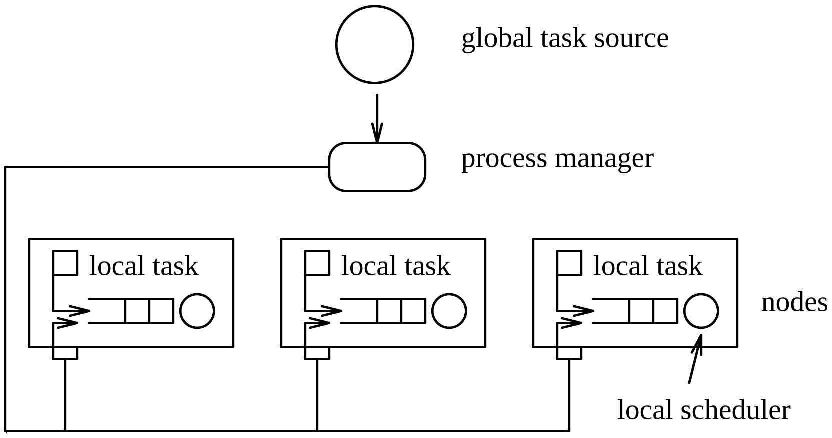

Figure 2: The System Model.

$$pex(X) \quad = \quad \textit{predicted} \text{ execution time of } X.$$

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:

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

Furthermore, if X is a simple subtask, we denote $node(X)$ as the execution node (or component) of X. That is, X is destined to be executed at $node(X)$.

3.2 The System Model

Our model of a distributed real-time system consists of a number of *nodes* representing different processing components of the system (Figure 2). These nodes manage different resources like database, expert system, number crunching computation, etc. Even the communication network is considered as one or more of the resources and is subsumed as one or more of the processing

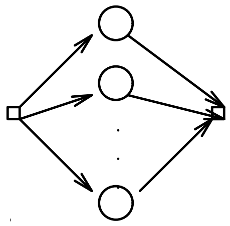

Figure 3: A global task that consists only of simple subtasks executing in parallel.

nodes. For example, a direct link between two sites is considered as one resource, while a LAN is considered another. 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 the schedulers decision are the real-time attributes associated with each task submitted to them.

Newly created global tasks are first processed by the process manager. The major functions of the process manager is to assign deadline 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. The resource requirement of the process manager is charged to the tasks it manages. We do not model this requirement explicitly.

4 The Parallel Subtask Problem (PSP)

In this section, we take a closer look at the parallel subtask problem (PSP). We first identify the problem and then suggest some possible solutions. These solutions will be evaluated using simulation (Section 5), and the results of the analysis will be presented in Section 6.

To study PSP, we only consider (in this section) global tasks of the form: $T = [T_1||T_2||...||T_n]$ where the T_i's $(1 < i < n)$ are all *simple subtask* (See Figure 3). For the global task T to meet its deadline $(dl(T))$, all T_i's 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. Missing a deadline occasionally is, although undesirable, possible. 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. As a quick example, if an average node in the system misses 5% of the tasks' deadlines, then the probability that a global task of 6 parallel

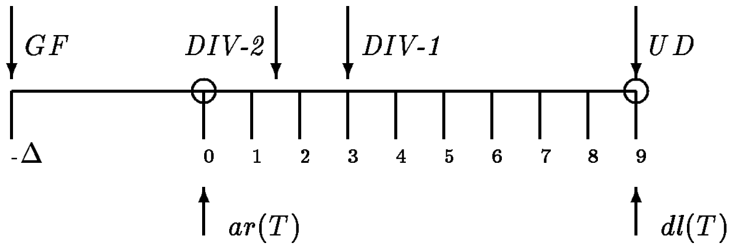

Figure 4: Example on the various PSP strategies applied to $T = [T_1||T_2||T_3]$.

subtasks misses his would be $1 - (1 - 0.05)^6 = 26.5\%$.[1] Comparing to a local task, or one that involves execution at only one node, a global task consumes more system "effort" to complete and yet it faces a much higher miss rate.[2] In firm real-time systems where tardy tasks are of no value and the resource invested in them wasted, the problem of missing global tasks simply because "they are big" is particularly costly. We will shortly look at several heuristics with varying degree of aggressiveness in helping global tasks to meet their deadlines.

4.1 Heuristics for PSP

To give global tasks a better chance of completing, we can assign their subtasks *virtual* deadlines before they are submitted to their execution nodes. These virtual deadlines are usually earlier than the tasks' *real* deadlines to give the nodes a sense of urgency of finishing the subtasks.

With our example global task $T = [T_1||T_2||...||T_n]$, our goal is to set a virtual $dl(T_i)$ from $dl(T)$. Figure 4 illustrates the various strategies listed below. In the example, time is relative to the arrival of T. The deadline of T is time 9.

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):

$$\textbf{UD:} \quad dl(T_i) = dl(T).$$

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

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

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

[1]Here, we assume that all the nodes in the system have similar miss rate, tasks have similar real-time constraints, and the queueing time at the nodes are independent.

[2]In this paper, we use the terms "miss rate" and "missed deadline probability" interchangeably.

earlier are the virtual deadlines assigned to the subtasks, and thus the higher the priority of the subtasks. As shown in Figure 4, *DIV-1* assigns a deadline of (9-0)/(3*1) + 0 = 3 to $dl(T_i)$, while for *DIV-2*, it is 9/6+0 = 1.5.

One issue with the *DIV-x* strategy concerns task abortion. If the local schedulers abort tasks whose (virtual) deadlines have already passed, the idea of pushing the virtual deadlines of subtasks early runs the risk of having the subtasks aborted by the schedulers. Going back to the example shown in Figure 4. Suppose the execution time of the subtasks are all 4 time units. If *DIV-1* is used, the virtual deadlines of the subtasks will be set to time 3 (see Figure 4). In this case, all three subtasks will be aborted at time 3 when the scheduler figures out that their (virtual) deadlines have already been missed. For the time being, we focus on non-aborting systems. Task abortion and its impact on the strategy performance is further discussed in Section 7.

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). To implement GF on a pure earliest-deadline-first scheduler, we subtract a big number Δ from the deadline of a global task and take the result as the subtasks' virtual deadline:

$$\textbf{GF:} \quad dl(T_i) = dl(T) - \Delta.$$

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 Simulation Model

To study the performance of the PSP strategies proposed in last section, we simulate the task and system models discussed in Section 3 with the various PSP strategies implemented into the process manager. This section describes the specifics of our simulation model.

As mentioned before, our soft real-time system consists of a number of nodes representing different processing components of the system. In general, these nodes can have vastly different characteristics: They may use different real-time scheduling policies (e.g., earliest-deadline first for CPU scheduling, shortest-seek-time first for disk scheduling, etc.) and task service time would follow different statistical distributions. If we model all this complexity, our results will be obscured by many intricate factors which impair our understanding of the basic tradeoffs of the PSP strategies. Instead, we chose to model a relatively simple and homogeneous system so that the observations made are more comprehensible.

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.

Our simulation model follows the basic construct as described in Section 3 (see Figure 2) with the following specification.

Nodes: There are k nodes in the system. Each node services their tasks using the *earliest-deadline-first*[3] scheduling algorithm.

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 a global task T consists of n subtasks $T_1, T_2, ..., T_n$ to be executed in parallel at n *different* nodes (we use the same value n for all global tasks). The execution times of the subtasks all follow the same exponential distribution with mean equal to $1/\mu_{subtask}$ time units. The rate of work due to global tasks is therefore $n\lambda_{global}/\mu_{subtask}$. Slack of global tasks is also uniformly distributed in the range $[S_{min}, S_{max}]$. The deadline of a global task is set by the following formula:

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

where $\max_i\{ex(T_i)\}$ is the execution time of the longest subtask among the T_i's, and $slack$ is the 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. This is because,

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

Overload Management Policy	No Abortion
Local Scheduling Algorithm	Earliest Deadline First
$\mu_{subtask}$	1.0
μ_{local}	1.0
k (# of nodes)	6
n (# of subtasks of a global task)	4
$load$	0.5
$frac\_local$	0.75
$[S_{min}, S_{max}]$	[1.25,5.0]

Table 1: Baseline setting

for a subtask T_j, we have,

$$sl(T_j) = dl(T) - ex(T_j) - ar(T) = \max_i\{ex(T_i)\} + slack + ar(T) - ex(T_j) - ar(T) \geq slack. \quad (3)$$

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,

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

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, i.e.,

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

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 simple 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 simple *subtasks*. However, since it takes n subtasks to make a global task, when $frac\_local$ is 1/2, there are n times more *local tasks* generated than *global tasks*.

Table 1 shows the parameter setting of our baseline experiment. 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. Finally, we note that although real-time systems should operate at a light load with few missed deadlines, it is the occasional experience of transient overload that accounts for most of the missed deadlines, testing the system's resiliency to emergency situations. We will therefore study the various deadline assignment strategies in an intermediate to high load environment, to highlight their performance differences.

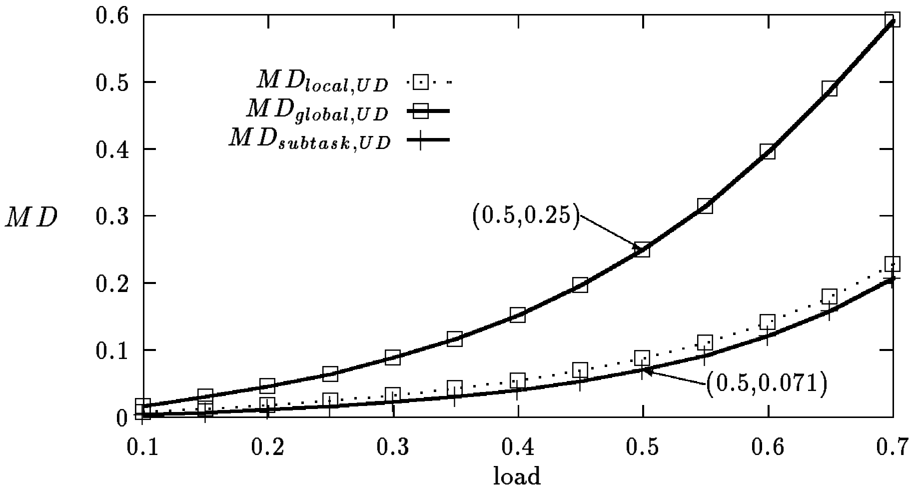

Figure 5: Performance of UD in baseline experiment.

6 Results

In this section, we show and discuss the results obtained from our simulation experiments. We will compare the performance of the various PSP strategies based on how well they can meet task deadlines.

To aid our discussion, we use the notation MD_A^B (or $MD_{A,B}$ in graph) to represent the *fraction of missed deadlines* of task type A under PSP strategy B. For example, $MD_{global}^{DIV1} = 0.4$ means that 40% of the global tasks miss their deadlines when *DIV-1* is the PSP strategy employed.

6.1 Baseline Experiment

UD. Figure 5 shows the performance of UD in our baseline experiment. 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. The MD of three task types are shown for local tasks (dotted □), simple subtasks of global tasks (+), and global tasks (solid □) (Recall that a global task consists of 4 simple subtasks to be executed in parallel).[4] As the load increases, the waiting time of tasks increases and more tasks (of all kind) miss their deadlines.

[4] In this paper, we will consistently use dotted lines for MD_{local} and solid lines for MD_{global} with different point styles representing different strategies.

Comparing local tasks and simple subtasks, we notice that simple subtasks have a slightly higher chance of *meeting* their deadlines (the + line is lower than the dotted □ line). This is due to the way we generate global task deadlines. As discussed in Section 5, even though global tasks have the same average slack as local ones, their subtasks, on average, have slightly more slack than locals' (see Equation 3 on page 11). This accounts for the difference between $MD_{subtask}^{UD}$ and MD_{local}^{UD}.

For a global task to meet its deadline, all four of its subtasks have to be finished by the deadline. From Figure 5, if the load of the system is 0.5, the probability that a subtask misses its deadline is about 7.1%. If subtasks miss their deadlines independently,[5] we would expect about $1-(1-7.1\%)^4 \simeq 25.5\%$ of the global tasks miss their deadlines. This number is not far from what we obtained from our experiment (25%), and is about 3 times as much as MD_{local}^{UD} (8.9%). 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 Equation 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.

Figure 6 compares the performance of UD (□) and *DIV-x* for $x = 1$ (◇) and $x = 2$ (×). Let us first focus on UD and *DIV-1*. 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. For example, at load 0.5, MD_{local} goes up from 9% to 11.7% while MD_{global} is reduced almost by half from 25% to 13%.

The relatively small increment on MD_{local} resulted from a switch from UD to *DIV-1* may be accounted for by the fact that only 25% of the workload is due to global tasks ($frac\_local =$ 0.75). The disturbance made to local tasks by subtasks with heightened priority is therefore mild. One may argue that since the majority of the work is due to local tasks, and the miss rate of locals is increased using *DIV-1*, will we end up missing more local tasks than the number of globals saved? The answer is yes, i.e., the overall *number* of missed deadlines is actually higher

[5] They don't. For example, if every global task generates subtasks for every node in the system, and there are no local tasks, then the subtask missed deadline probability will be highly co-related. We make the independent assumption here just to illustrate the problem.

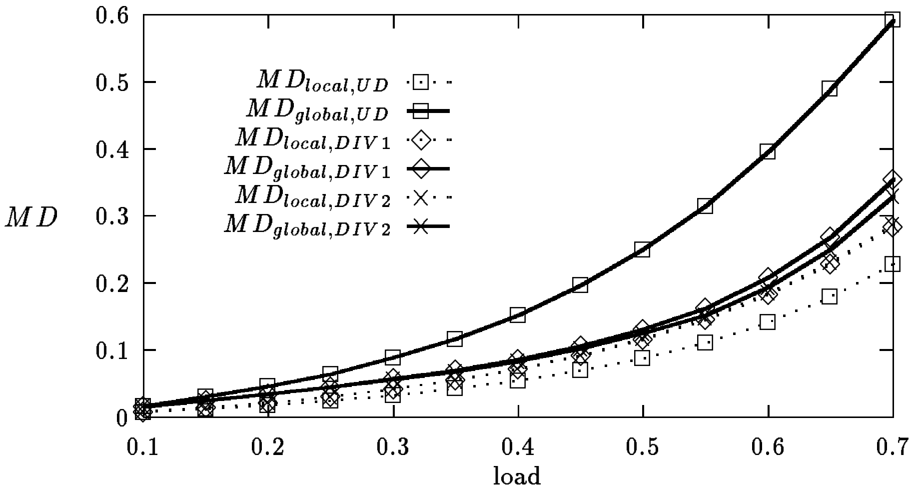

Figure 6: Performance of UD and *DIV-x* in baseline experiment.

for *DIV-1* than UD. However, if we count the work done on tardy tasks as the amount of missed work, *DIV-1* does have an advantage in this category. In fact, at a load of 0.5, our experiment shows that the fraction of missed work is reduced from $0.75 \times 0.09 + 0.25 \times 0.25 = 0.13$ to $0.75 \times 0.117 + 0.25 \times 0.13 = 0.12$. Recall that our main goal is to lower the unacceptably high global task miss ratio. In this respect, as Figure 6 shows, *DIV-1* is quite an effective strategy for the baseline setting. We will look at the effect of having a different job mix on the PSP strategies later.

Figure 6 also shows MD_{local} and MD_{global} for *DIV-2* ($\times$). 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. We will further discuss the question of how to set the value of x for the *DIV-x* strategy in a later section.

GF. The minute difference between *DIV-1* and *DIV-2* as shown in Figure 6 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. This is shown in Figure 7. Figure 7 is basically the same as Figure 6 except that we replace the curves for *DIV-2* ($\times$) by those of GF ($\triangle$).

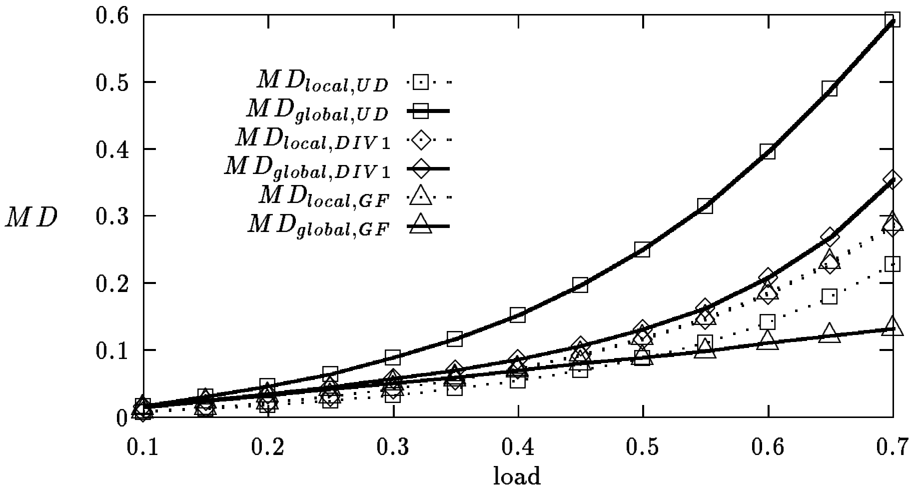

Figure 7: Performance of *UD*, *DIV-1*, and *GF* in baseline experiment.

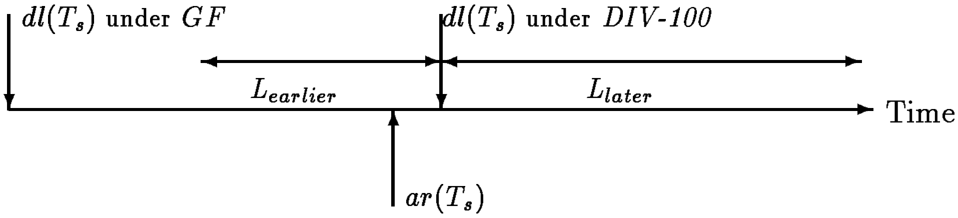

Figure 8: Queueing position of T_s under *DIV-100* and *GF*

Comparing *DIV-1* with *GF* in Figure 7, we notice that both of them miss approximately the same number of local tasks while *GF* misses significantly fewer number of global tasks than *DIV-1* does, particularly under high load situation. To understand how *GF* can meet more global task deadlines without losing on the local side, let us situate ourselves as a particular subtask, T_s, just submitted to a node, i.e., $ar(T_s)$ = current time. We are interested in comparing the queueing position of T_s under *DIV-x* and *GF* strategies. From Figure 6, we see that the performance difference between *DIV-1* and *DIV-2* is quite small (and as will be shown later, the performance of *DIV-x* flattens out as x increases). To make the contrast more dramatic, we will consider *DIV-100* (or any x that is excessively big). Now, under *DIV-100*, T_s will see two types of local tasks in the waiting queue of the node: those with deadlines later than $dl(T_s)$ and those with earlier deadlines.[6] We call these two sets of locals L_{later} and $L_{earlier}$ respectively. Figure 8 illustrates the queueing position of T_s. Loosely speaking, since we use a very big x (100) in *DIV-x*, $dl(T_s)$ is set very close to $ar(T_s)$, which is also the current time. Any local tasks in the set $L_{earlier}$ would thus have deadlines that are either very close to the current time, or

[6]Here, $dl(T_s)$ is the virtual deadline assigned to T_s.

which have already been past. In other words, these are the tasks that are very likely to miss the deadlines. Now, by switching from *DIV-x* to GF, we are setting the priority of T_s higher than any locals. T_s is therefore cutting the line and gets itself ahead of any locals in $L_{earlier}$. We make three observations over this switch of priority:

1. Queueing time of locals in L_{later} is not affected.
2. Queueing time of locals in $L_{earlier}$ is lengthened.
3. Queueing time of T_s is shortened.

Since the locals in $L_{earlier}$ are going to miss their deadlines no matter whether *DIV-100* or GF is used, prolonging their queueing time will not affect the miss rate of local tasks. On the other hand, shortening the queueing time of subtasks (such as T_s's) does significantly reduce the miss rate of global tasks. Furthermore, the higher the system load is, the more local tasks will there be in $L_{earlier}$, and the queueing time for the subtasks will be reduced further. This explains why MD^{GF}_{global} is much smaller than MD^{DIV1}_{global} especially under high load.

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.

7 More on *DIV-x* and GF

Now that we have shown that *DIV-x* and GF are two viable solutions for PSP, in this section, we study when they are most effective. In particular, we will study the impact of the following factors on their performance: (1) The value of x for *DIV-x*. (2) The relative proportion of local tasks in the system. (3) Abortion on tardy tasks. (4) Global tasks with different number of subtasks.

7.1 Choosing x

In last section, we compared *DIV-1* and *DIV-2* and showed that the performance difference is not significant. So, is *DIV-x* sensitive to the value of x? Referring to the *DIV-x* formula:

$$dl(T_i) = [dl(T) - ar(T)]/(n * x) + ar(T),$$

we see that two important factors are affecting the value of the virtual deadlines set to the subtasks: (1) the number of subtasks (n) of a global task, and (2) the value of x. The larger the

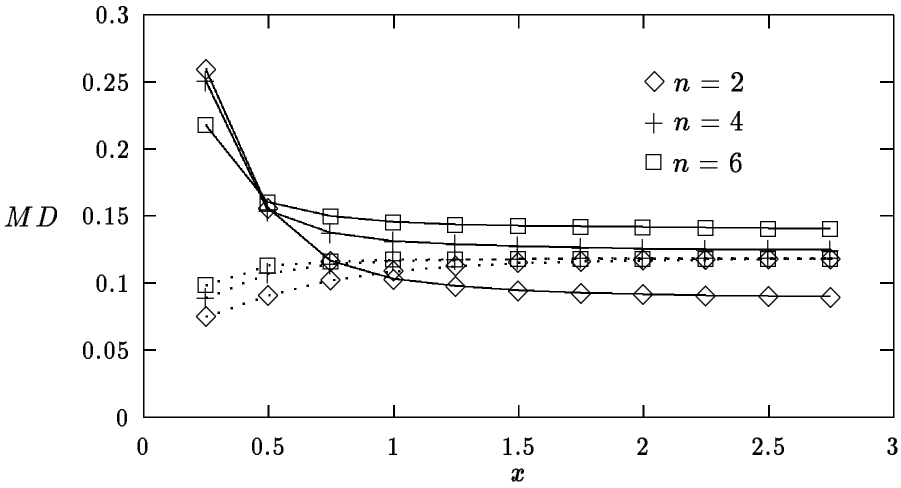

Figure 9: MD^{DIVx} as functions of x under different value of n. (Dotted lines for MD_{local}; Solid lines for MD_{global}.)

product $n * x$ is, the smaller is the virtual deadline and thus the higher is the subtask priority. This, in turn, results in a lower MD_{global} and a higher MD_{local}. Now, if n is small (e.g., 2), a global task has only a small number of subtasks and PSP is not a serious problem. Therefore, we may not need a very large x or a large boost in subtask's priority to keep MD_{global} low. On the other hand, if n is large, even though PSP is serious, the $n * x$ product is already large, and again, we do not need a really big x. In fact, through extensive experiment, we find out that setting $x = 1$ is usually adequate. As an example, Figure 9 shows MD_{global}^{DIVx} and MD_{local}^{DIVx} as a function of x for the cases where the number of subtasks of a global task (n) is 2, 4, and 6.

From Figure 9, we see that all MD curves flatten out as x increases. The values of MD_{global} and MD_{local} stablize at a smaller value of x as n becomes larger. Since 2 is the smallest number of subtasks that a global task can ever have, and the curves for $n = 2$ ($\diamond$) almost reach their stablized points when $x = 1$, setting $x = 1$ is therefore sufficient.[7]

7.2 Local Task Population

In our baseline experiment, we set $frac\_local$, the fraction of work in the system that is due to local tasks to 75%. The performance of GF and *DIV-x* depends on this parameter. For example, if there are no local tasks in the system (i.e., $frac\_local = 0$), GF will perform exactly the same as UD because the deadlines of all subtasks are reduced by exactly the same amount (Δ) by

[7]There is no harm for setting x larger for the no-abortion case, but see Section 7.3 for the abortion case.

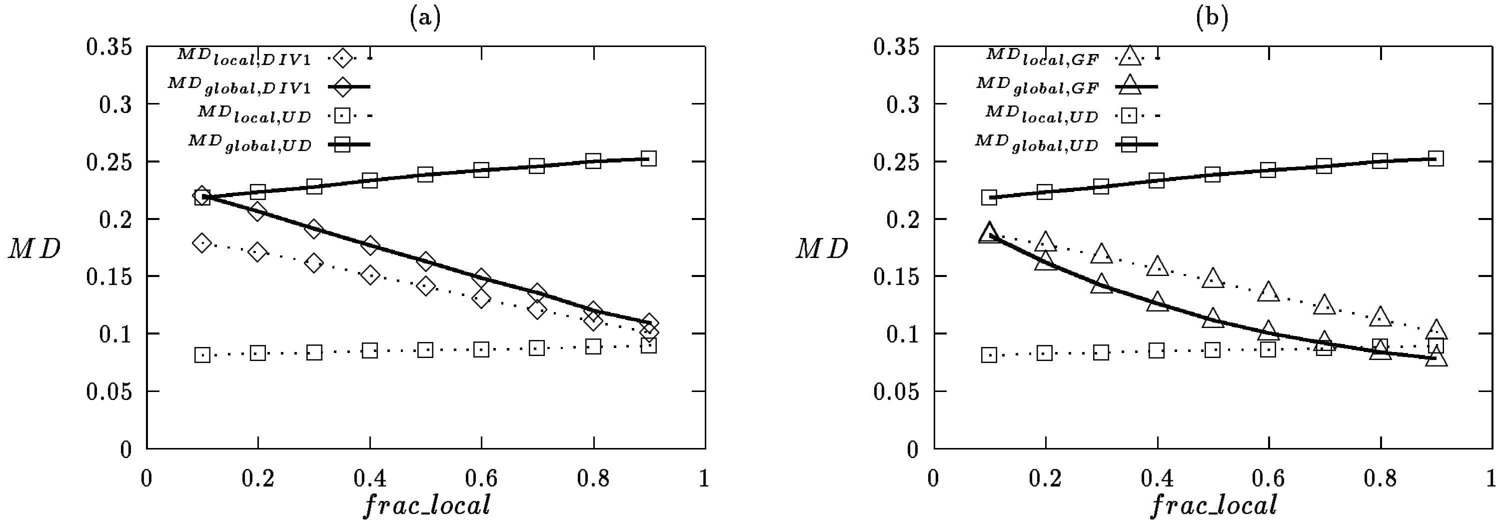

Figure 10: Performance of (a) *DIV-1* and (b) GF as functions of *frac_local*.

GF. Figures 10(a) and 10(b) show MD^{DIV1} and MD^{GF} as functions of *frac_local* respectively. MD^{UD} is also shown for comparison.

From the figures, we see that as *frac_local* increases, both MD^{UD}_{local} and MD^{UD}_{global} increase slightly. This is due to the way we generate global task deadlines. Again, a global task deadline is generated as the arrival time plus a random slack plus the execution time of the *longest* subtask (see Equations 2 and 3 in Section 5). In other words, on average, the deadline of a global task is slightly larger than that of a local that is generated at the same time. Therefore, global tasks have slightly lower priority than locals and are thus slightly less competitive. As a result, as *frac_local* increases, more *competitive* local tasks are introduced to the system replacing some *less competitive* global tasks, and higher MD values are observed for both classes of tasks.

On the other hand, when either *DIV-x* or GF is used, the subtasks are given early virtual deadlines. This raises their priority and makes them *more competitive* than local tasks. MD^{DIVx} and MD^{GF} thus drop as *frac_local* increases. As shown in Figure 10, *DIV-x* and GF are most effective when the system has a relatively large population of local tasks.

7.3 Abortion

We consider two different ways of aborting tardy tasks: (1) abortion by process manager and (2) abortion by local scheduler. For case (1), we assume that the local schedulers do not initiate any abortion. They will keep working on a task even though its (virtual) deadline has already

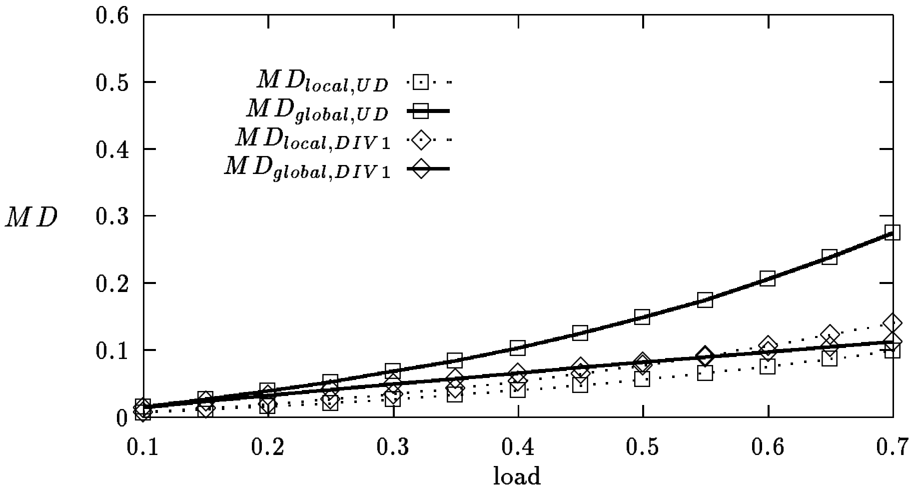

Figure 11: Performance of UD and *DIV-1* in baseline experiment with abortion.

expired. We implement this type of abortion using timers. When a task (global or local) is generated, an associated timer is set to run off at the task's *real* deadline. If the task is not finished before the timer expires, it is aborted[8]. Figure 11 shows the performance of UD and *DIV-1* when process manager abortion is implemented.

Comparing Figure 7 and Figure 11, we see that abortion helps reduce all miss rates by not wasting resources on tardy tasks. *DIV-1* again is very effective in keeping MD_{global} low (e.g., at $load = 0.5$, $MD^{UD}_{global} = 15.0\%$, while $MD^{DIV1}_{global} = 7.8\%$). The performance of GF in this case is very similar to *DIV-1* and whose curves are omitted from the graph for legibility.

For the second type of abortion, namely, abortion by local schedulers, a subtask is aborted if its virtual deadline is missed. A local scheduler thus takes an active role and aborts tasks based on the deadlines presented to it. This causes a serious problem in the implementation of GF and *DIV-x*. Recall that under GF, the virtual deadline of a subtask is set by deducting a big number from its global task's deadline. The virtual deadline is thus always less than the arrival time of the subtask, and the subtask will be aborted right away. GF is therefore inapplicable in this situation. Similarly, *DIV-x* also pushes the virtual deadlines early. If x is set too big, the virtual deadlines will be too close to the arrival time and subtasks will be aborted before they finish even though their real deadlines are not expired.

Unnecessary abortions by local schedulers bring about two adverse effects. First, resource invested in aborted subtasks is wasted. Second, an aborted subtask has its slack consumed

[8] For a global task, all its subtasks are aborted.

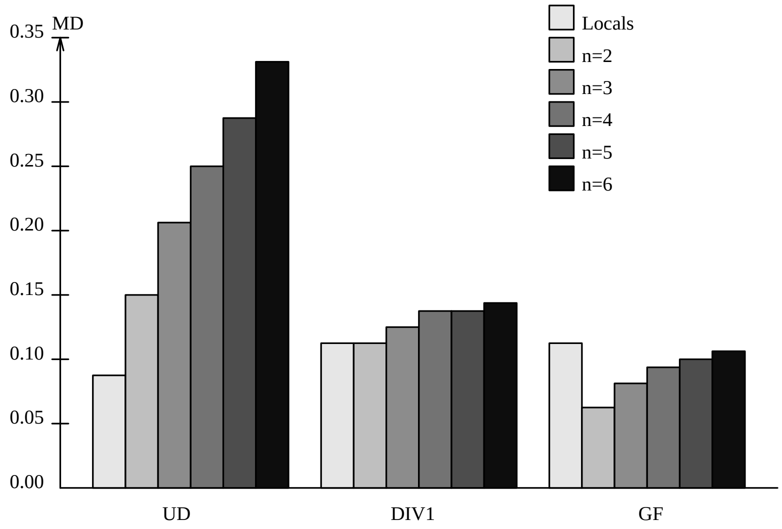

Figure 12: MD of different task classes under the various PSP strategies.

mostly by its former unsuccessful trial. The subtask thus has little slack left when it is resubmitted and will very likely miss its deadline. Results of experiments not shown here indicate that *DIV-x* performs poorly with local scheduler aborts. This is particularly so in a moderate to tight environment (i.e., high load, small slack). In this situation, special directives should be given to nodes specifying that subtasks are non-abortable locally.

7.4 Non-homogeneous Global Tasks

In our baseline experiment, we assume that all global tasks consist of exactly four parallel subtasks, and the execution time of the subtasks are all generated from the same exponential distribution. In this subsection, we generalize our base model to cover a non-homogeneous system. Due to space limitation, here we only consider systems in which global tasks may have different number of subtasks.

We modified our baseline experiment such that the number of subtasks of a global task is chosen uniformly from the range [2..6]. There are thus six classes of tasks (locals + 5 classes of globals). Figure 12 shows the fraction of missed deadlines of each class of task under different PSP strategies.

As expected, UD has trouble meeting the deadlines of global tasks, especially those of the *large* ones. For example, a global task with 6 subtasks has an one-third chance of missing its deadline, and which is about 4 times as likely as that of local tasks. On the other hand, at the expense of missing some more local tasks, *DIV-1* manages to keep the MD of all task classes at roughly the same level. Finally, by hoisting the priority of global tasks, GF further reduces global tasks miss rates to even lower values.

This concludes our discussion on the parallel subtask problem. We will look at the serial subtask problem next.

8 SSP + PSP

In this section, we will briefly discuss the serial subtask problem and mention some of the results presented in [6]. We will also show how to integrate our PSP and SSP strategies.

To recapitulate, SSP concerns global tasks that consist of a number of *serial* stages, or subtasks. Consider a global task $T = [T_1 T_2 ... T_n]$. The ultimate deadline ($dl(T)$) fails to represent the tightness of each individual subtask. For example, if T_1 is scheduled with the deadline $dl(T)$, the scheduler will consider the time that should be reserved for the other subtasks as slack to T_1. Task T_1 will be running at a low priority because of its excessive slack. As a remedy, earlier virtual deadline should be assigned so as to reserve enough amount of time for the subtasks to follow.

In [6], several ways of breaking up an end-to-end deadline into intermediate virtual deadlines that can better reflect the urgency of each subtask is studied. Here, we consider one of them, called *Equal Flexibility* (EQF). In words, EQF tries to estimate the total amount of *slack* that a global task has and divides this slack among the subtasks proportional to their execution times. Each subtask thus has the same slack-to-execution-time ratio (flexibility). Using our example, before a subtask T_i is submitted for execution, EQF assigns a virtual deadline to T_i according to the following formula:[9]

$$dl(T_i) = ar(T_i) + pex(T_i) + \overbrace{\underbrace{[(dl(T) - ar(T_i) - \sum_{j=i}^{m} pex(T_j))]}_{\text{total amount of slack left}} * \underbrace{[pex(T_i)/\sum_{j=i}^{m} pex(T_j)]}_{\text{pex() fraction}}}^{\text{slack assigned to } T_i}.$$

Note that EQF requires an estimate on task execution time. This estimation, however, need not be very accurate. As shown in [6], EQF delivers good performance even when the estimate can be off by a factor of 2.

The performance of EQF is studied extensively using the same model discussed in Section 3. It is shown that EQF significantly reduces serial global task miss rate over the policy of assigning the ultimate deadline as subtask deadline (UD). This improvement is particularly

[9]Recall that ar() and pex() represent the arrival time and the predicted execution time of a task respectively.

```
FUNCTION SDA(X:task; D:deadline)
BEGIN
        IF (X is a simple subtask) then
           dl(X) := D
        ELSE IF X = [X_j X_{j+1}...X_n] THEN
           assign virtual deadline to task X_j according to the SSP strategy.
           SDA(X_j, dl(X_j));
        ELSE IF X = [X_1||X_2||...||X_n] THEN
           PAR i = 1 FOR n
                assign virtual deadline to X_i according to the PSP strategy.
                SDA(X_i,dl(X_i));
        ENDIF
END
```

Figure 13: SDA algorithm for integrated SSP and PSP.

marked in cases when global tasks have (1) a non-trivial number of subtasks (e.g. > 3), and (2) sufficient amount of slack (e.g. when the miss rate of globals under UD is less than 50%).

The PSP strategies discussed in this paper and the SSP strategies (such as EQF) 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. Figure 8 shows an algorithm which breaks down an end-to-end deadline (D) of a global task (X) into sub-deadlines for the *executable* simple subtasks[10].

To study the relative importance of the SSP and PSP strategies and how they affect complex distributed tasks, we ran an experiment with different SSP/PSP strategies applied to a specific class of serial-parallel global tasks. Table 2 shows the deadline assignment combinations, and Figure 14 shows the task graph of the global tasks generated in this experiment. Each global task has 5 serial stages. Stages 2 and 4 are complex subtasks each with 4 parallel simple subtasks. Using our stock trading application example, the 5 stages can represent (1) initialization, (2) distributed information gathering, (3) analysis, (4) action implementation, and (5) conclusion of the trading task.

The experiment follows the same baseline setting as shown in Table 1 except that global tasks are replaced by those shown in Figure 14 and the slack of global tasks is chosen uniformly from the range [6.25,25]. This slack distribution is equal to that of local task ([1.25,5]) scaled up by 5 (because our globals have 5 stages). Figure 15 shows the MD of locals and globals

[10] A subtask is executable if it is not preceded by any other.

SDA	SSP	PSP
UD-UD	UD	UD
UD-DIV1	UD	DIV1
EQF-UD	EQF	UD
EQF-DIV1	EQF	DIV1

Table 2: Combination of SSP/PSP strategies.

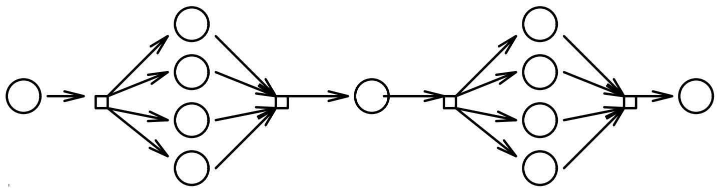

Figure 14: Task graph

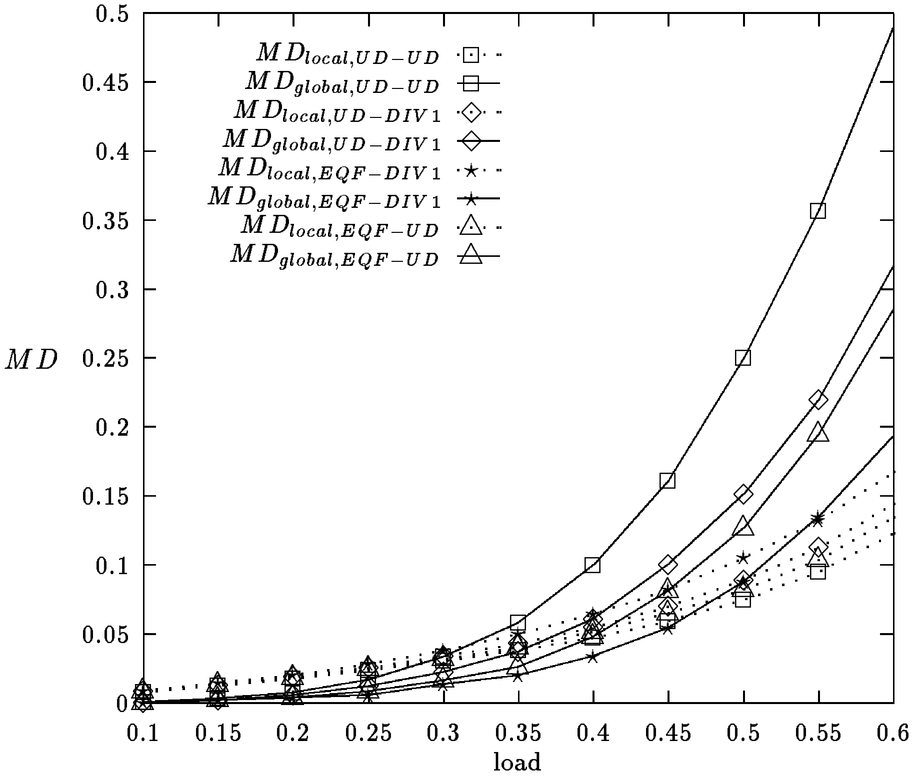

Figure 15: Performance of different SDA strategies.

with different SDA strategies applied.

From the figure, we see that under low load, global tasks have a slightly lower miss rate. This is because of their much larger slack. However, as load increases, the *UD-UD* strategy ($\square$) misses vastly more global deadlines than it misses local ones. The application of either EQF ($\triangle$) or *DIV-1* ($\diamond$) significantly reduces MD_{global} with a mild increment in MD_{local}. Nonetheless, they are not quite adequate if applied singularly, particularly under high load situation. The two strategies compliment each other and together ($\star$) they are able to keep MD_{global} close to MD_{local}, even up to a load of 0.6 in our experiment.

Although the results we presented in this section is limited to one particular type of serial-parallel task, they do clearly suggest 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.

9 Conclusion

This paper studied the problem of assigning deadlines to subtasks of complex distributed global tasks in a soft real-time environment. We showed that a single end-to-end global deadline often fails to represent the right priority of individual subtasks. This results in a very large global task miss rate. As curative measures, two different algorithms for assigning virtual deadlines to subtasks were presented and evaluated. These virtual deadlines are earlier than the end-to-end global deadlines so as to speed-up the progress of global tasks. Our results showed that a good subtask deadline assignment strategy can significantly reduce global task miss rate without severely jeopardizing local tasks.

For the class of serial-parallel tasks, the SDA problem can be divided into two sub-problems: SSP and PSP. While EQF is a good strategy for SSP [6], this paper showed that two strategies: *DIV-x* and GF are quite effective for PSP. These two strategies are most outstanding under high load situation and when there is a non-trivial population of local tasks in the system. Between *DIV-x* and GF, GF usually holds an edge if tardy task abortion 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.

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 I/O requests with deadlines: a performance evaluation. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 113–124, 1990.

[3] 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.

[4] S. Cheng, J. A. Stankovic, and K. Ramaritham. Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–174, 1986.

[5] C. Hou and K. G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 146–155, 1992.

[6] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437, 1993.

[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*, 1992.

[11] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 108–115, 1990.

[12] K. Ramaritham, J. A. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, 1990.

[13] 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.

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