Deadline Assignment in Distributed Hard Real-Time Systems with Relaxed Locality Constraints

Jan Jonsson *

Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
janjo@ce.chalmers.se

Kang G. Shin

Real-Time Computing Laboratory
Dept. of Elec. Engr. and Computer Science
University of Michigan, Ann Arbor, MI 48109
kgshin@eecs.umich.edu

Abstract

In a real-time system, tasks are constrained by global end-to-end deadlines. In order to cater for high task schedulability, these deadlines must be distributed over component subtasks in an intelligent way. Existing methods for automatic distribution of end-to-end deadlines are all based on the assumption that task assignments are entirely known beforehand. This assumption is not necessarily valid for large real-time systems. Furthermore, most task assignment strategies require information on deadlines in order to make good assignments, thus forming a circular dependency between deadline distribution and task assignment. We present a heuristic approach that performs deadline distribution prior to task assignment. The deadline distribution problem is presented in the context of large distributed hard real-time systems with relaxed locality constraints, where schedulability analysis must be performed off-line, and only a subset of the tasks are constrained by predetermined assignments to specific processors. Using experimental results we identify drawbacks of previously-proposed techniques, and then show that our solution provides significantly better performance for a large variety of system configurations.

1 Introduction

In a distributed real-time computing system, applications are decomposed into tasks, which are then assigned to processors according to *locality constraints* that are either *strict* (the assignment of a task is known beforehand) or *relaxed* (there exist more than one assignment alternative for each task). There are some well-known solutions to the real-time task assignment problem, but an important remaining problem is *deadline distribution*. To guarantee the functionality of a real-time system, an application task is constrained to start its execution and complete within a given time span called the *end-to-end deadline*. Tasks have usually been logically decomposed into a set of sequential and/or parallel *subtasks*, often because the system designers are forced to modularize software for maintainability and reusability reasons or exploit parallelism for performance reasons. As a consequence of this decomposition, the end-to-end deadline must be distributed over the component subtasks, *i.e.*, a local deadline needs to be assigned to each subtask for its scheduling. The local deadlines should be assigned to each of the subtasks in an intelligent way that allows the subtasks to be scheduled locally on each processor as efficiently as possible. One good strategy for deadline distribution is to adopt a divide-and-conquer approach that first divides the overall problem into smaller problems that are solved locally and then combined to obtain a global solution, thus reducing the complexity of the problem.

Many researchers have addressed the deadline distribution problem [1, 2, 3, 4, 5, 6, 7], all under a common assumption that task assignments are entirely known, *i.e.*, strict locality constraints. In many real-time systems, however, only a small number of task assignments are governed by strict locality constraints, *e.g.*, those tasks constrained by demands of resources in their physical proximity such as sensors and actuators. The constraints on the remaining task assignments are not strict and the assignments can be made using such techniques as clustering heuristics [8], critical path heuristics [9], list scheduling heuristics [10], or branch-and-bound algorithms [11]. Deadline distribution using conventional techniques can, therefore, be performed only if the task assignment is completely known. Task assignment techniques, on the other hand, require information about individual task deadlines for scheduling purposes. Thus, there exists a circular dependency between the deadline distribution and task assignment problems. Moreover, conventional deadline distribution techniques can only produce as good results as provided for by the initial task assignment. Given a poor task assignment

*This work was done while the author was at the Real-Time Computing Laboratory, Dept. of Elec. Engr. and Computer Science, University of Michigan, Ann Arbor. He was supported in part by a grant from the Volvo Research Foundation and the Volvo Educational Foundation.

0-8186-7813-5/97 $10.00

as input, the resulting deadline distribution may be just as bad. Finding the most suitable task assignment is unfortunately an NP-complete problem [12] and, therefore, good solutions to the combined deadline distribution and task assignment problem must be found through the use of sub-optimal heuristic algorithms.

In this paper, we solve the deadline distribution problem using sub-optimal heuristic strategies for both deadline distribution and task assignment. Our solution is based on the *Basic Slicing Technique (BST)* in [1], but deadline distribution is performed *prior* to the task assignment phase. The deadline distribution problem is addressed in the context of distributed hard real-time systems with relaxed locality constraints. In such systems, task assignment and scheduling are usually assumed to be performed off-line in order to guarantee the 100% *a priori* schedulability of each hard real-time task in the system. Systems with these characteristics are mission/safety-critical where the workload is known beforehand, but the number of tasks is too large for each task assignment to be specified in advance.

Our main contributions in this paper are:

i) We show in what aspects the BST algorithm and metrics are inadequate for the assumed system. This is achieved by means of an experimental evaluation in which BST is used for an application consisting of a randomly-generated task graph and a multiprocessor system of varying size. This evaluation enables us to identify shortcomings of the BST metrics that will reveal themselves when applied to systems where parallelism in the application cannot be fully exploited.

ii) We propose an improvement of BST, called the *Adaptive Slicing Technique (AST)*, that encompasses a set of new metrics that are capable of adapting themselves to changes in workload and system size. We demonstrate how AST yields substantially better results than BST for the given system, especially in minimizing the maximum task lateness.

The rest of the paper is organized as follows: Section 2 describes related work on deadline distribution. Section 3 describes the assumed task model. Section 4 describes the deadline distribution problem and presents a basic algorithm to solve the problem. Section 5 describes the experimental setup. Section 6 presents the experimental evaluation of BST. Section 7 introduces AST and demonstrates its improved performance over BST. Section 8 discusses complementary results and possible future work. Finally, Section 9 summarizes the results in this paper.

2 Related work

The technique (BST) proposed by Di Natale and Stankovic [1] assigns slices, execution windows with static positions in time, to subtasks using a critical path concept. The strategy used for finding slices is to determine a critical path in the task graph that maximizes the minimum laxity of the tasks. Two laxity ratio metrics (previously proposed in [6, 7]) were used for evaluating paths in the task graph: one assigns a subtask deadline based on its execution time, and the other assigns a subtask deadline based on the number of subtasks in the critical path. The slicing technique is optimal in the sense that it maximizes the minimum task laxity in the application. However, optimality applies only if task assignment is completely known in advance. The technique was demonstrated using a non-preemptive time-triggered run-time model, but is not inherently constrained to such a run-time model.

The technique proposed by Abdelzaher and Shin [2] specializes on improving a given task assignment where local deadlines have already been assigned. Using a branch-and-bound strategy, an optimal solution is found in acceptable time as long as the system workload is kept below a certain limit. The subtask execution windows have dynamic positions in time rather than static as in [1], which caters for better exploitation of processor resources. The technique is optimal in the sense that it minimizes the maximum task lateness in the application, but only under the assumptions that task assignment is known beforehand, a non-preemptive time-triggered run-time model is employed, and real-time communication channels have adjustable priorities. The usefulness of this technique has yet to be demonstrated for systems in which these assumptions do not hold.

In [3], Gutiérrez García and González Harbour proposed a heuristic iterative approach that, given an initial local deadline assignment, finds an improved solution in reasonable time. For each iteration a new deadline assignment is calculated based on a metric that measures by "how much" schedulability failed. Bettati and Liu [4] presented a technique for scheduling a system of flow-shop tasks. Local deadlines are assigned by distributing end-to-end deadlines evenly over subtasks. For this method, the simplifying assumption is made that execution times are either identical for all subtasks or identical for all subtasks assigned to the same processor. Saksena and Hong [5] proposed a deadline distribution technique based on a critical scaling factor that is applied to the subtask execution times. The end-to-end deadline is expressed as a set of local deadline assignment constraints. Given a set of local deadline assignments, they calculated the largest value of the scaling factor that still makes the subtasks schedulable. The local deadline assignment is then chosen to maximize the largest value of the scaling factor. The techniques in [3, 4, 5] all assume that tasks consist of purely sequential subtasks and that task assignment is known beforehand.

Kao and Garcia-Molina presented multiple strategies for distributing end-to-end deadlines over sequential [6] and sequential/parallel [7] subtasks. However, these strategies are only aimed at, and evaluated in the context of, soft real-time systems with complete *a priori* knowledge of task-processor assignment.

3 Task model

A real-time application is composed of one or more *tasks*, each representing a major computation. Each task consists of a set of *subtasks* (modules) that represent smaller (than the task itself) entities of computation. The task set that realizes a given application can be represented by a directed acyclic *task graph*, in which the nodes represent subtasks and the arcs represent precedence constraints between subtasks. A subtask τ_i is a *predecessor* of another subtask τ_j if τ_j cannot begin its execution until τ_i has completed its execution. Conversely, subtask τ_j is called the *successor* of subtask τ_i. A subtask which has no predecessors is called an *input subtask*, whereas a subtask which has no successors is called an *output subtask*.

Subtask τ_i is characterized by a tuple $\langle c_i, r_i, d_i \rangle$ where c_i is the *execution time*, a worst-case estimate of the computational demands of the subtask; r_i is the *release time*, the earliest time at which the subtask is allowed to start its execution; and d_i is the *relative deadline*, the amount of time within which the subtask must complete its execution once it has been released. The *absolute deadline* D_i of the subtask is the sum of the subtask's release time and relative deadline. A *path* is a sequence of subtasks in the task graph for which the first and the last subtasks are regarded as a subtask pair $\langle \tau_1, \tau_n \rangle$ subject to an *end-to-end deadline* D that specifies the maximum time measured from the release of subtask τ_1 to the completion of subtask τ_n.

The activities associated with message transfer from subtask τ_i to subtask τ_j are handled by a *communication subtask* χ_{ij} characterized by the tuple $\langle m_{ij}, r_{ij}, d_{ij} \rangle$ where m_{ij} denotes the maximum message size, r_{ij} the message release time, and d_{ij} the relative deadline of the message. The real communication cost for sending a message depends on the communication scheduling strategy employed in the system and cannot be determined until the subtasks have been assigned to processors.

Note that we only need to assume non-periodic tasks for the purposes of this paper. For an application with periodic tasks we can always transform the original periodic tasks into a set of non-periodic tasks that execute within an interval $[0, L)$, where L is the least common multiple of the periods of all periodic tasks involved. Thus, we allow precedence constraints and communication between subtasks of tasks with different periods.

4 Deadline distribution

4.1 Problem statement

Given an end-to-end deadline D and a corresponding subtask pair $\langle \tau_1, \tau_n \rangle$, the *deadline distribution problem* is to partition (distribute) D into release time r_i and relative deadline d_i for each subtask τ_i in the task graph in such a way that the constraint $d_1 + d_2 + \cdots + d_n \leq D$ is satisfied for each path Φ between τ_1 and τ_n.

A solution to the deadline distribution problem cannot be accepted simply because it satisfies the above condition. One also has to consider the practical issue of schedulability: the relative deadline of a subtask must be derived in such a way that the subtask is likely to be feasibly scheduled. This can be achieved by assigning an ample *slack* (difference between relative deadline and execution time) to each subtask so that it can meet its absolute deadline even in the presence of contention with other subtasks for the processor. The slack for a path Φ is defined as the difference between the end-to-end deadline D_Φ and the accumulated execution time of all subtasks in the path.

Two important metrics that are often used in the evaluation of a deadline distribution strategy is the *laxity* and the *lateness* of a subtask. The laxity of a subtask is the maximum amount of time that the execution of the subtask can be delayed without missing its absolute deadline. A subtask's laxity is determined before the subtask is scheduled and is thus an indicator on how much contention for the processor the subtask can withstand during scheduling. The lateness of a subtask is the difference between the completion time of the subtask and its absolute deadline, *i.e.*, a non-positive quantity for valid schedules. A subtask's lateness is determined after the subtasks have been scheduled and is an indicator on the quality of the schedule.

Here, we will assess the performance of a deadline distribution strategy by its capability to minimize the maximum task lateness. This performance metric refers to the lateness of only one subtask, and is thus an indicator on "how far" from infeasibility the schedule is and how much additional background workload the schedule can handle.

4.2 Basic algorithm

Our deadline distribution algorithm also uses the concept of critical path. A critical path in a task graph is the one that optimizes a given metric. Correct identification of a critical path is crucial for the quality of the deadline distribution and the system's schedulability. When a critical path has been identified, the end-to-end deadline is distributed over the subtasks in the critical path. For a system with complete *a priori* information on task-processor assignment and interprocessor communication cost, the best critical path can easily be found as described in [1]. When the assignment is not entirely fixed, however, finding the best critical path is no longer an easy task. The reason

```
1.  initialize set Π with all subtasks in the task graph;
2.  while { Π ≠ ∅ } loop
3.    find a critical path Φ in Π that minimizes metric R;
4.    distribute the end-to-end deadline of Φ by assigning
        release times and deadlines to the subtasks in Π;
5.    for { each subtask τ in Π } loop
6.      for { each predecessor τ_p of τ } loop
7.        assign an end-to-end deadline to τ_p that
            is equal to the release time of τ;
8.      end loop;
9.      for { each successor τ_s of τ } loop
10.       assign a release time to τ_s that
            is equal to the absolute deadline of τ;
11.     end loop;
12.   end loop;
13.   remove all subtasks in Φ from Π;
14. end loop;
```

Figure 1: Basic deadline assignment algorithm.

is that it is not yet known what pairs of subtasks will be afflicted with interprocessor communication overhead. Therefore, the deadline distribution algorithm must rely on the prediction of the "possibly best" critical path. Deadline distribution techniques such as the one in [1] might not be appropriate for systems without *a priori* knowledge of task-processor assignment. Namely, if the prediction as to where the critical path is turns out to be incorrect, one may assign insufficient amounts of slack to those subtasks that constitute the real critical path in the final schedule.

A basic algorithm for distributing end-to-end deadlines over the subtasks is given in Figure 1. The algorithm takes a task graph as the input and produces an annotated task graph containing information about subtask release times and relative deadlines. Both ordinary subtasks and communication subtasks are considered in the algorithm, implying that communication subtasks also have release times and deadlines assigned to them. This allows for the use of deadline-based communication scheduling strategies such as the one in [13]. The algorithm in Figure 1 is similar in structure to that for the Basic Slicing Technique (BST) in [1]. However, the algorithm in Figure 1 is able to handle systems with relaxed locality constraints because communication subtasks will be processed by the algorithm regardless of whether their corresponding communication costs are known or not. For this, the algorithm contains a communication cost estimation phase in Step 3. The various steps of the algorithm are described below in detail.

Initialize subtask set (Step 1): We assume that all input and output tasks have already been assigned appropriate release times and end-to-end deadlines, respectively, according to the temporal requirements of the application. All subtasks in the task graph are then inserted into a subtask set Π that represents all subtasks not yet assigned release times and deadlines. Recall that communication subtasks are also inserted in Π at this stage.

Find a critical path (Step 3): The breadth-first traversal of the task graph determines a critical path Φ among all potential paths for the subtasks in Π. Ties among paths with identical metric values are broken arbitrarily. Evaluation metrics R suitable for our purpose will be discussed in Section 6 and Section 7. As mentioned earlier, communication subtasks are taken into account when identifying a critical path, regardless of whether the real communication cost is known or not. So, one must estimate the communication cost when it is not known. Suitable communication cost estimation strategies will be discussed in Section 5.

Distribute the end-to-end deadline (Step 4): The end-to-end deadline D_Φ of the critical path Φ found in Step 3 is distributed over the subtasks in Φ. The deadline distribution is governed by the constraint that the release time of a subtask must be equal to the absolute deadline of its predecessor in Φ. Thus, all subtasks in the path will be assigned *slices*, non-overlapping execution windows, of the end-to-end deadline. Only those communication subtasks whose communication cost, real or estimated, is non-negligible will be assigned an execution window. Note that, whereas the execution time windows of subtasks in the same path cannot overlap, the execution windows of subtasks in different paths may overlap and thus are subject to contention for available processors during scheduling.

Attach the remaining subtasks (Step 5–Step 11): The subtasks in Φ now constitutes a "spine" to which the remaining subtasks must attach, *i.e.*, adapt their release times and absolute deadlines. Therefore, the release time for each subtask not in Φ is set to the latest absolute deadline of any predecessor subtask in Φ. Similarly, the absolute deadline for each subtask not in Φ is set to the earliest release time of any successor subtask in Φ. This absolute deadline now becomes a new end-to-end deadline in the graph.

Remove critical-path subtasks (Step 13): The subtasks in Φ are removed from Π to mark that they have been assigned release times and deadlines.

Repeat until no subtasks remain (Step 2): The main loop in the algorithm is repeated until no subtasks are left in Π.

5 Experimental setup

5.1 System architecture

We have used an experimental platform based on a homogeneous multiprocessor architecture with a shared bus interconnection network. The system size ranges from 2 to 16 processors. We assumed that the shared bus is time-multiplexed in such a way that the communication cost between two processors is one time unit per transmitted data item. Communication between two subtasks residing on

the same processor is done via accessing shared memory and its cost assumed to be negligible. We also assumed that communication in the network can take place concurrently with processor computation.

5.2 Workload

In all experiments[1], a set of 128 task graphs were generated using a random task graph generator. Each task graph contained between 40 and 60 subtasks. Subtask execution times were chosen at random assuming a uniform distribution with a mean execution time (MET) of 20 time units. To mimic different distributions of subtask execution times, three scenarios were assumed in the simulations. For the first scenario, *low distribution execution time (LDET)*, the subtask execution times deviated by at most $\pm 25\%$ from the mean execution time. For the second scenario, *medium distribution execution time (MDET)*, the execution time deviation was at most $\pm 50\%$, and for the third scenario, *high distribution execution time (HDET)*, the execution time deviation was at most $\pm 99\%$. An end-to-end deadline was chosen for each input–output subtask pair in such a way that the *overall laxity ratio (OLR)* between the end-to-end deadline and the accumulated task graph workload corresponded to 1.5. The number of successors/predecessors to each subtask was chosen at random to be in the range of 1 to 3, and the depth of the task graph was chosen at random in the range of 8 to 12 levels. The number of data items in each message passed between a pair of subtasks was chosen in such a way that the *communication–to–computation cost ratio (CCR)* between the average message communication cost and the average subtask execution time corresponded to 1.0.

5.3 Task assignment algorithm

The tasks were scheduled using a deadline-driven version of the list scheduling algorithm in [10]. For each scheduling step, the list scheduler selected one subtask from all schedulable subtasks (those whose predecessors have been scheduled) using an earliest-deadline-first policy. Then, the subtask was scheduled on the processor that yielded the earliest start time for the subtask assuming a non-preemptive time-driven run-time scheduling model. The chosen strategy allows us to reproduce BST results in [1] for systems with strict locality constraints, and thus provides a basis for a fair evaluation.

5.4 Communication cost estimation strategies

As mentioned in Section 4.2, it is not clear how interprocessor communication cost should be estimated when task assignments are not known beforehand. In Step 3 of the algorithm in Figure 1, this is resolved by using a communication cost estimation strategy whenever a communication subtask with unknown cost is encountered. In our experiments, we will evaluate two strategies for estimating the communication cost between communicating subtasks: the *Communication Cost Non-Existing (CCNE)* strategy which assumes that there will never be interprocessor communication between subtasks, and the *Communication Cost Always Assumed (CCAA)* strategy which assumes that there always will be interprocessor communication.

[1] All modeling and simulation in the experiments were performed within FEAST [14], a framework for evaluation of allocation and scheduling techniques for distributed hard real-time systems.

6 The Basic Slicing Technique (BST)

We conducted a set of experiments where we evaluate two deadline distribution metrics presented for the Basic Slicing Technique (BST) in [1]. The purpose of the experiments is to identify drawbacks with BST when it is applied to a system with relaxed locality constraints, justifying the need of a new technique.

Two metrics are defined for the BST in [1]. The first, the *normalized laxity ratio (NORM)*, is the ratio of available path slack to the sum of the execution times of all subtasks in a path Φ:

$$R_{NORM} = (D_\Phi - \sum_{\tau_i \in \Phi} c_i) / \sum_{\tau_i \in \Phi} c_i$$

With this metric, the relative deadline for subtask τ_i will be $d_i = c_i(1 + R_{NORM})$, *i.e.*, slack is assigned in proportion to subtask execution time. The second metric, the *pure laxity ratio (PURE)*, is the ratio of available path slack to the number of subtasks n_Φ in a path Φ:

$$R_{PURE} = (D_\Phi - \sum_{\tau_i \in \Phi} c_i) / n_\Phi$$

With this metric, the relative deadline for subtask τ_i is defined as $d_i = c_i + R_{PURE}$, *i.e.*, all subtasks are assigned an equal share of slack.

The plots in Figure 2 summarize the results attained when the generated task graphs were scheduled using the algorithm in Figure 1. The plots show the maximum subtask lateness as a function of system size for each choice of communication cost estimation strategy. The task lateness shown in the plots is the average of the maximum task lateness taken over the 128 simulation runs that were made for each parameter combination. Recall that more negative values on lateness means better performance. The plots to the left, middle and right in each figure correspond to the LDET, MDET, and HDET scenarios, respectively.

As can be seen from Figure 2, both metrics show poor performance for small system sizes. This is because the contention between subtasks over a few available processors forces multiple subtasks to be scheduled within overlapping execution windows. As the system size increases, the maximum subtask lateness will decrease almost linearly with increasing system size until a point at which lateness saturates. The task assignment algorithm can exploit more parallelism in the task graph, thereby decreasing the length of critical paths in the graph. With shorter

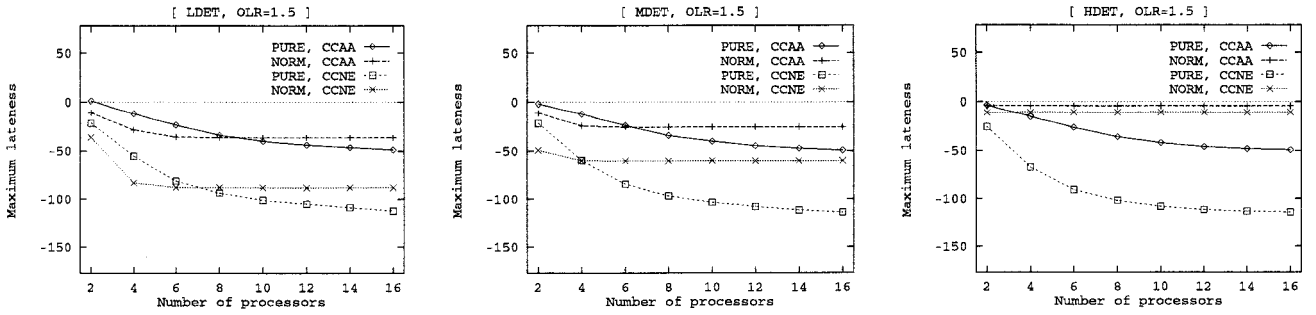

Figure 2: Maximum task lateness for the PURE and NORM metrics.

critical paths, there will be more overall slack to distribute among the subtasks. However, as soon as the graph parallelism is fully exploited, nothing more can be gained by increasing the system size and hence flattening the curves for larger system sizes.

The overall best performance is attained when the communication cost is never assumed (CCNE) because the maximum overall slack will be available for distribution over the subtasks. When the subtasks are subsequently scheduled, any interprocessor communication between subtasks will only consume slack from the receiving subtask. When the communication cost is always assumed (CCAA), on the other hand, precedence constraints in the graph will consume slack from the overall slack pool since these constraints will be modeled as communication subtasks with an estimated non-zero communication cost. Each such communication subtask will thus reduce the amount of slack available for the subtasks in the same path as the communication subtask.

The overall best metric is PURE, because it is relatively insensitive to the amount of variation in execution time. For NORM, on the other hand, performance will be drastically degraded when the variation in execution time increases because the availability of subtasks with short execution times will also increase. Since NORM assigns less slack to subtasks with shorter execution times, the maximum lateness will increase when the availability of short subtasks increases.

7 The Adaptive Slicing Technique (AST)

Based on the observations on BST, we can now lay the foundation for the Adaptive Slicing Technique (AST), the improved deadline distribution technique. We design AST so that it assumes no communication cost overhead and distributes deadlines according to an equal-share strategy, thus resulting in high performance with respect to PURE, the best BST metric demonstrated in Section 6. In this section, we propose two improved metrics that reduce the negative effects of the PURE metric. We then evaluate these metrics together with the PURE metric and show that the insufficiencies have to a large extent been eliminated.

As discussed in Section 6, PURE combined with the CCNE strategy will perform poorly when task graph parallelism cannot be fully exploited on the system, because subtasks will contend for the few available processors and slack will be consumed by interprocessor communication for receiving subtasks. Because PURE employs an equal-share slack distribution strategy, subtasks with longer execution times will be the most vulnerable to processor contention and slack consumption. Therefore, we will introduce two concepts as an aid in remedying these problems, namely, *execution time threshold* and *virtual execution time*. The execution time threshold is a mechanism for guaranteeing only certain subtasks to be allotted extra slacks. By using the execution time threshold to filter out subtasks with large enough execution times, we can improve the performance with respect to PURE in those situations where task graph parallelism cannot be fully exploited. The purpose of a virtual execution time is to make a subtask appear computationally more consuming than it actually is. By assigning a virtual execution time to a subtask larger than the real execution time, the deadline distribution algorithm will allocate more slack to that subtask if its real execution time is above the given execution time threshold. A similar threshold strategy is briefly discussed in [1]. However, no attempt is made in [1] to determine appropriate threshold levels or to evaluate the effect of the threshold strategy on schedule quality.

We now introduce the *threshold laxity ratio (THRES)* metric, which is similar to the PURE metric but with a virtual execution time c' instead of the real execution time c. The virtual execution time for subtask τ_i is defined as

$$c'_i = \begin{cases} c_i & \text{if } c_i < c_{thres} \\ c_i(1+\Delta) & \text{if } c_i \geq c_{thres} \end{cases}$$

where c_{thres} is the execution time threshold and Δ is a *surplus factor* that defines the amount of slack by which the

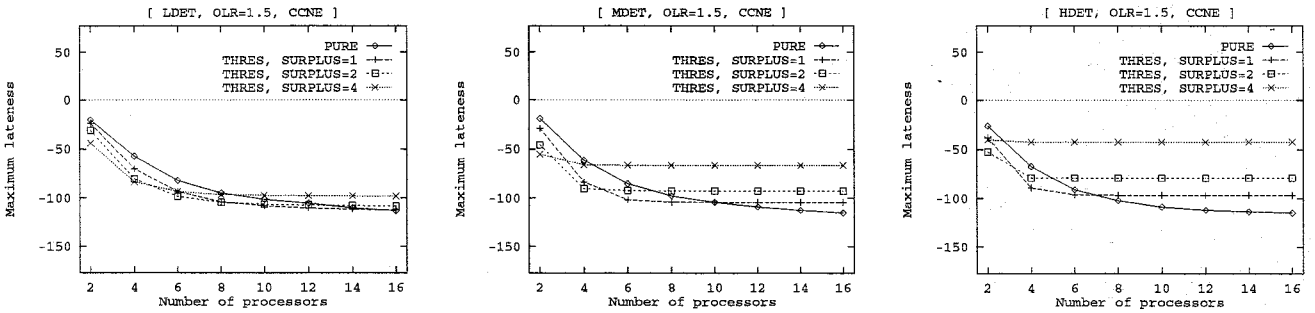

Figure 3: Maximum task lateness for different choices of surplus factor for the THRES metric.

real execution time should be increased for those subtasks whose execution times exceed c_{thres}.

Figure 3 illustrates how the choice of surplus factor Δ affects the maximum subtask lateness in the final schedule assuming the values 1, 2 and 4 of Δ, respectively. When parallelism can be fully exploited, a low Δ-value is preferable because there will be little contention between subtasks for the available processors and as much slack as possible should be distributed evenly over the subtasks. In fact, too large a value of Δ can be detrimental to the performance as can be seen for the case of $\Delta = 4$. As the number of processors decreases and full exploitation of task graph parallelism is no longer possible, it is important to assign extra slack to subtasks with large execution times in order to handle processor contention. Thus, a larger value of Δ is needed to cater for the assignment of extra slack to subtasks with large execution times. Because of the counteracting effects on the maximum task lateness that occur when the surplus factor is changed, a "best" value of Δ is nigh impossible to find. Therefore, a trade-off must be made in the choice of Δ so that acceptable performance is attained for both small and large systems. Unfortunately, the trade-off may have to be made with a certain workload assumed, a restriction that to some extent limits the applicability of the THRES metric in a general context.

Figure 4 illustrates how the choice of execution time threshold affects the maximum subtask lateness in the final schedule when c_{thres} is based on the mean execution time (MET) of all subtasks in the graph. As can be seen in the plots, the performance improves as the threshold increases. However, the differences are not that significant: for instance, when c_{thres} is varied $\pm 25\%$ around the mean execution time as in Figure 4, the performance does not vary more than $\pm 5\%$. Thus, the choice of execution time threshold is not as critical as the choice of surplus factor. However, it is recommended that the threshold be kept at a value close to the mean execution time in order to make the metric useful.

It is, as have been observed in Figure 3, very hard to attain a consistent high performance for THRES with a fixed value of the surplus factor Δ. We therefore propose an improvement of the threshold-based metric wherein the amount of assigned surplus is not fixed, but will adapt itself to the degree of task graph parallelism that can be exploited. The resulting metric, which we call the *adaptive laxity ratio (ADAPT)*, is similar to the THRES metric but with a different definition of the virtual execution time:

$$c'_i = \begin{cases} c_i & \text{if } c_i < c_{thres} \\ c_i(1 + \xi/N_{proc}) & \text{if } c_i \geq c_{thres} \end{cases}$$

where ξ is the average task graph parallelism and N_{proc} is the number of processors in the system. The average task graph parallelism is defined as the total task graph work load divided by the length, in execution time, of the longest path in the graph.

By assigning slack based on the ratio of the average task graph parallelism to the number of processors, we are able to compensate for the low performance exhibited for the PURE metric when task graph parallelism cannot be fully exploited. Because the number of processors is in the denominator of the surplus factor expression, ADAPT will follow the behavior of PURE as more parallelism is exploited when the system size increases. This can be observed in Figure 5 where results for the PURE, THRES and ADAPT metrics are presented. For the simulations, we used a surplus factor $\Delta = 1$ for THRES, and an execution time threshold c_{thres} for THRES and ADAPT that were 25% higher than the subtask MET. Figure 5 shows that, for small systems where task graph parallelism cannot be fully exploited, the ADAPT metric clearly outperforms the THRES and PURE metrics. In these cases, the increase in performance over PURE can be as high as 100%. As the system size increases, the performance of ADAPT will be comparable to that of PURE. The THRES metric also performs quite well for small systems, but will exhibit lower performance than PURE as the system size increases. From these observations, we can conclude that the

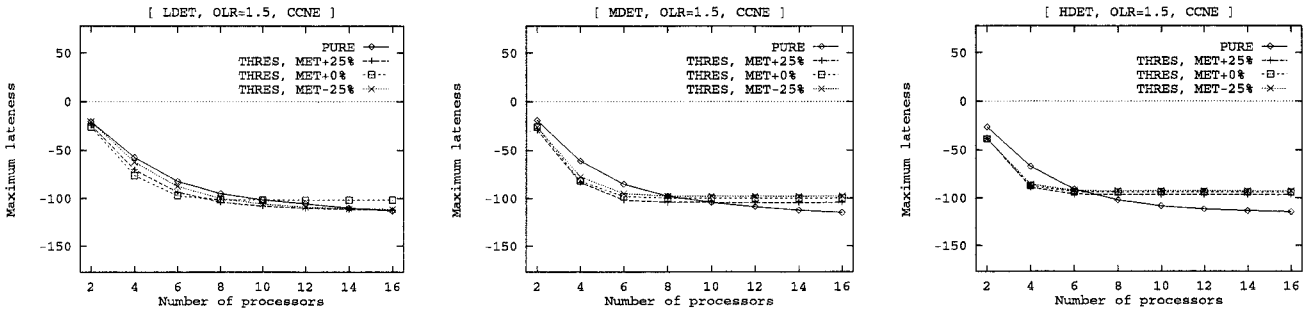

Figure 4: Maximum task lateness for different choices of execution time threshold for the THRES metric.

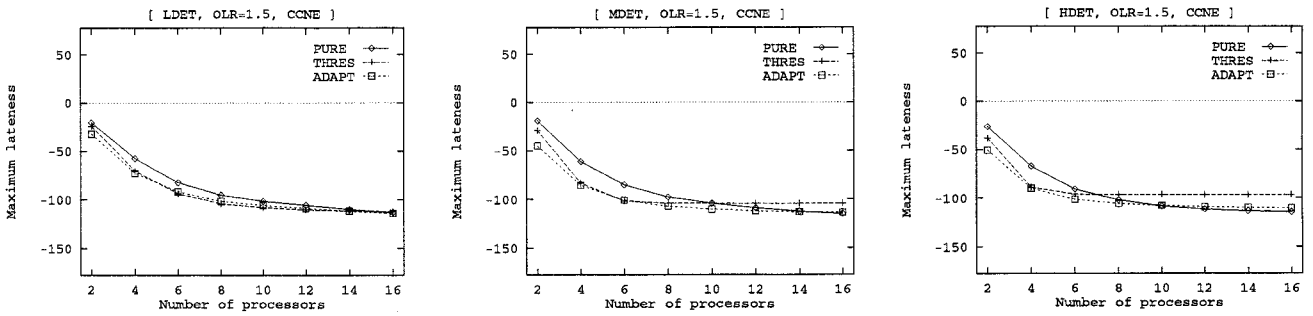

Figure 5: Maximum task lateness for the THRES and ADAPT metrics.

adaptive ingredient of the ADAPT metric functions well for the assumed system.

ADAPT is not entirely without flaws, however, as can be seen for the HDET scenario. When the number of processors exceeds 10, the lateness saturates and becomes slightly worse than for the PURE metric. This is because the existence of an execution time threshold does not permit as good slack distribution for larger systems, an effect that is clearly demonstrated in Figure 4. Also, the surplus factor expression still makes a small contribution to the virtual execution time, yielding less slack to distribute. Since the variation in subtask execution time is larger for the HDET scenario than for the other scenarios, the effect will be more noticeable here as there is a higher availability of subtasks with long execution times.

8 Discussion

Due to space limitation, we have omitted the results of some complementary experiments. The full results of these experiments can be found in [15], but we will briefly summarize the results here. We have evaluated AST for task graphs with varying degrees of parallelism and with both larger and smaller mean subtask execution times. Using the same basic experimental setup as described in Section 5, we found that AST scales very well with these parameters when the ADAPT metric is used. For the THRES metric, on the other hand, experimental results confirmed that a universally best value for the surplus factor is hard to find. The "best" value of the surplus factor is very application-dependent and must in the worst case be chosen specifically for each system. We have also evaluated AST for different interconnection topologies and values on the communication–to–computation cost ratio (CCR). Here, too, did we find that AST scales well with different parameter values.

In this paper we have studied task graphs with random structures. However, we would also like to investigate the performance of AST for commonly-encountered structures such as in-tree, out-tree, and fork-join task graphs. In addition, we would like to evaluate AST on a set of realistic benchmarks that do not only encompass small comprehensible applications like the one presented in [1], but also larger applications. Furthermore, we would like to make measurements on systems that utilize contention-based communication scheduling techniques for real-time channels [13]. It is far from obvious how the communication cost for a real-time channel should be estimated in a system with relaxed locality constraints when no or only a few real-time channels are known beforehand. Although AST has been evaluated under a time-driven non-

preemptive scheduling policy, neither AST nor BST is restricted to that policy. Therefore, we would also like to explore the quality of AST under various task assignment and scheduling policies. Moreover, because the results here are based on a homogeneous processing architecture, the applicability of AST on a heterogeneous system is also worthy of further investigation.

Finally, some comments on the computational complexity of AST are in order. Although the optimality that follows with BST will have to be renounced due to the relaxed locality constraints in the assumed system, AST will still inherit the highly tractable time complexity of BST. Since the complexity of AST only differs from that of BST by a constant factor (the extra passes through the task graph for calculating the accumulated workload and finding the longest path), the overall time complexity of the AST algorithm is $O(n^3)$ for a task composed of n subtasks.

9 Conclusions

Distribution of end-to-end deadlines over subtasks in a distributed real-time system is an important, but difficult, problem to solve. It is particularly difficult to solve the problem for systems with relaxed locality constraints where a majority of the subtasks are not pre-assigned to particular processors. Many solutions have been proposed for the problem, but only for systems with strict locality constraints where task assignment is entirely known beforehand. In this paper, we have presented a heuristic solution, called the Adaptive Slicing Technique (AST), to the deadline distribution problem for systems with relaxed locality constraints. AST is a refinement of the Basic Slicing Technique (BST) [1] in the sense that it is circumventing difficulties associated with deadline distribution over subtasks with an unknown initial task assignment. Using new metrics for minimizing the maximum task lateness, a set of experiments using random task graphs demonstrated that AST performs well in situations where BST has been shown to exhibit poor performance. For example, the increase in performance of AST over BST can be as high as 100% for small system sizes when parallelism in the application cannot be fully exploited. At the same time, AST performs at least as good as BST in all other situations.

References

[1] M. Di Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, Dec. 7–9, 1994, pp. 216–227.

[2] T. F. Abdelzaher and K. G. Shin, "Optimal Combined Task and Message Scheduling in Distributed Real-Time Systems," *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 5–7, 1995, pp. 162–171.

[3] J. J. Gutiérrez García and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems," *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Santa Barbara, California, Apr. 25, 1995, pp. 124–132.

[4] R. Bettati and J. W.-S. Liu, "End-to-End Scheduling to Meet Deadlines in Distributed Systems," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Yokohama, Japan, June 9–12, 1992, pp. 452–459.

[5] M. Saksena and S. Hong, "An Engineering Approach to Decomposing End-to-End Delays on a Distributed Real-Time System," *Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Honolulu, Hawaii, Apr. 15–16, 1996, pp. 244–251.

[6] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, Pittsburgh, Pennsylvania, May 25–28, 1993, pp. 428–437.

[7] B. Kao and H. Garcia-Molina, "Subtask Deadline Assignment for Complex Distributed Soft Real-Time Systems," *Proc. of the IEEE Int'l Conf on Distributed Computing Systems*, Poznan, Poland, June 21–24, 1994, pp. 172–181.

[8] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, Apr. 1995.

[9] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, May 1996.

[10] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger, "Multiprocessor Scheduling with Interprocessor Communication Delays," *Operations Research Letters*, vol. 7, no. 3, pp. 141–147, June 1988.

[11] D.-T. Peng and K. G. Shin, "Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-Time Systems," *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, New Port Beach, California, June 1989, pp. 190–198.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[13] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-Time Communication in Multihop Networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1055, Oct. 1994.

[14] J. Jonsson and J. Vasell, "Evaluation and Comparison of Task Allocation and Scheduling Methods for Distributed Real-Time Systems," *Proc. of the IEEE Workshop on Real-Time Applications*, Montreal, Canada, Oct. 21–25, 1996, pp. 226–229.

[15] J. Jonsson and K. G. Shin, "Deadline Assignment in Distributed Hard Real-Time Systems with Relaxed Locality Constraints," Technical Report No. 281, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden, Dec. 1996.